Database Management System

EE504A

Online Courseware (OCW)

B.TECH (3RD YEAR – 5th SEM)

Department of Electrical Engineering

Guru Nanak Institute of Technology
(Affiliated to MAKUT, West Bengal , Approved by AICTE - Accredited by NAAC – 'A+' Grade)
157/F Nilgunj road, Panihati, Kolkata-700114, West Bengal

Prerequisite:
1. Logic of programming language
2. Basic concepts of data structure and algorithms

Course Objectives
1. Tolearnthedata models,conceptualizeanddepicta databasesystem 2. To design system usingE-Rdiagram.
3. TolearnSQL&relationaldatabasedesign.
4.Tounderstandtheinternalstoragestructuresusingdifferentfileandindexingtechniques.
5. To know the concepts of transaction processing, concurrency control techniques and recovery procedure.

Course Outcomes (COs):
On successful completion of the learning sessions of the course, the learner will be able:

EE504A.1: To Express the knowledge of data models.
EE504A.2: To implement the concept of designing an efficient relational database system.
EE504A.3: To Correlate real world queries with database system. CS503.4: To Illustrate transaction processing, concurrency control and recovery management of a database.
EE504A.5: To Assess the internal storage structure to implement a proper database for an organization.

EE504A: Database Management System:
Contact Hours: 3L
Credits: 3
Lecture: 36L

Module 1:
Introduction [3L]
Concept & Overview of DBMS, Data Models, Database Languages, Database Administrator, Database Users, Three Schema architecture of DBMS.
Module 2:
Entity-Relationship and Relational Database Model [11L]
Basic concepts, Design Issues, Mapping Constraints, Keys, Entity-Relationship Diagram, Weak Entity Sets, Extended E-R features, case study on E-R Model. Structure of relational Databases, Relational Algebra, Relational Calculus, Extended Relational Algebra Operations, Views, Modifications of the Database.

Module 3:
SQL and Integrity Constraints [6L]
Concept of DDL, DML, DCL. Basic Structure, Set operations, Aggregate Functions, Null Values, Domain Constraints, Referential Integrity Constraints, assertions, views, Nested Subqueries, Database security application development using SQL, Stored procedures and triggers.

Module 4:
Relational Database Design [8L]
Functional Dependency, Different anomalies in designing a Database., Normalization using functional dependencies, Decomposition, Boyce-Codd Normal Form, 3NF, Normalization using multi-valued dependencies, 4NF, 5NF , Case Study

Module 5:
Internals of RDBMS [9L]
Physical data structures, Query optimization: join algorithm, statistics and cost based optimization. Transaction processing, Concurrency control and Recovery Management: transaction model properties, state serializability, lock base protocols; two phase locking, Dead Lock handling

Module 6:
File Organization & Index Structures [6L]
File & Record Concept, Placing file records on Disk, Fixed and Variable sized Records, Types of Single-Level Index (primary, secondary, clustering), Multilevel Indexes

Lesson Plan

Stream : EE
Subject : Database Management System
Subject Code : EE504A

Lecture No.	Description	Reference
1.	Concept & Overview of Database and database management system	1, 2, 3
2.	Data Models, Database Languages, Database Administrator, Database Users	1, 2, 3
3.	Three Schema architecture of DBMS	1, 2, 3
4.	Entity-Relationship Model, Basic concepts , Design Issues, Mapping Constraints, Keys	1, 2, 3,5
5.	Entity-Relationship Diagram, Weak Entity Sets	1, 2, 3,5
6.	Extended E-R features, Aggregation etc	1, 2, 3,5
7.	Structure of relational Databases, Relational Algebra, select operations on different relations	1, 2, Ref. 2
8.	Relational Algebra, different operations	1, 2, Ref. 2
9.	Relational Algebra problem solving	1, 2, Ref. 2
10.	Relational Calculus, Operations on different relations	1, 2, Ref. 2
11.	Tuple Relational calculus	1, 2, Ref. 2
12.	Domain Relational calculus	1, 2, Ref. 2
13.	Views, Modifications of the Database	1, 2, Ref. 2
14.	Concept of DDL, DML, DCL. Basic Structure	1, 2, Ref. 2
15.	Set operations, Aggregate Functions, Null Values	1, 2, Ref. 2
16.	Domain Constraints, Referential Integrity Constraints, assertions	1, 2, Ref. 2
17.	Views	1, 2, Ref. 2
18.	Nested Sub-queries	1, 2, Ref. 2
19.	Problem solving using SQL	1, 2, Ref. 2
20.	Database security application development using SQL	1, 2, Ref. 2
21.	Stored procedures and triggers.	1, 2, Ref. 2
22.	Functional Dependency.	1, 2, Ref. 3
23.	Different anomalies in designing a Database	1, 2, Ref. 3
24.	Normalization using functional dependencies,	1, 2, Ref. 3
25.	Decomposition, 1NF, 2NF	1, 2, Ref. 3

Lecture No.	Description	Reference
26.	Problem solving on 1NF, 2NF	1, 2, Ref. 3
27.	3NF, Boyce-Codd Normal Form	1, 2, Ref. 3
28.	Problem solving on 3NF, BCNF	1, 2, Ref. 3
29.	Normalization using multi-valued dependencies, 4NF, 5NF	1, 2, Ref. 3
30.	Physical data structures, Query optimization: join algorithm	1, 2, Ref. 2
31.	statistics and cost based optimization	1, 2, Ref. 2
32.	Transaction processing	1, 2, 4, Ref. 2
33.	Transaction model , ACID properties	1, 2, 4, Ref. 2
34.	Serializability	1, 2, 4, Ref. 2
35.	Concurrency control	1, 2, 4, Ref. 2
36.	Recovery Management	1, 2, 4, Ref. 1
37.	Lock based protocols	1, 2, 4, Ref. 1
38.	two phase locking	1, 2, 4, Ref. 1
39.	Dead Lock handling	1, 2, 4, Ref. 1
40.	File & Record Concept, Placing file records on Disk	1, 2, Ref. 4
41.	Fixed and Variable sized Records,	1, 2, Ref. 4
42.	Types of Single-Level Index (primary, secondary, clustering)	1, 2, Ref. 4
43.	Multilevel Indexes, B tree, B+ tree	1, 2, Ref.4

Text Books:

1. Henry F. Korth and Silberschatz Abraham, “Database System Concepts”, Mc.Graw Hill.
2. Elmasri Ramez and Novathe Shamkant, “Fundamentals of Database Systems”, Benjamin Cummings Publishing. Company.
3. Ramakrishnan: Database Management System , McGraw-Hill
4. Gray Jim and Reuter Address, “Transaction Processing : Concepts and Techniques”, Moragan Kauffman Publishers.
5. Ullman JD., “Principles of Database Systems”, Galgottia Publication.

Reference:

Ref. 1. Jain: Advanced Database Management System CyberTech
Ref. 2. Date C. J., “Introduction to Database Management”, Vol. I, II, III, Addison Wesley.
Ref. 3. “Fundamentals of Database Systems”, Ramez Elmasri, Shamkant B.Navathe, Addison Wesley Publishing Edition
Ref. 4. “Database Management Systems”, Arun K.Majumdar, Pritimay Bhattacharya, Tata McGraw Hill

MODULE 1: Introduction [3L]
Concept & Overview of DBMS, Data Models, Database Languages, Database Administrator, Database Users, Three Schema architecture of DBMS.

1.1 Concept & Overview of DBMS:

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data.

This is a collection of related data with an implicit meaning and hence is a database. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise.

The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*.

By **data,** we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. You may have recorded this data in an indexed address book, or you may have stored it on a diskette, using a personal computer and software such as DBASE IV or V, Microsoft ACCESS, or EXCEL.

A **datum** – a unit of data – is a symbol or a set of symbols which is used to represent something. This relationship between symbols and what they represent is the essence of what we mean by **information**. Hence, information is interpreted data – data supplied with semantics.

Knowledge refers to the practical use of information. While information can be transported, stored or shared without many difficulties the same cannot be said about knowledge. Knowledge necessarily involves a personal experience. Referring back to the scientific experiment, a third person reading the results will have information about it, while the person who conducted the experiment personally will have knowledge about it.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Because information is so important in most organizations, computer scientists have developed a large body of concepts and techniques for managing data. These concepts and technique form the focus of this book. This chapter briefly introduces the principles of database systems.

1.2 Data Processing Vs. Data Management Systems

Although Data Processing and Data Management Systems both refer to functions that take raw data and transform it into usable information, the usage of the terms is very different.

Data Processing is the term generally used to describe what was done by large mainframe computers from the late 1940's until the early 1980's (and which continues to be done in most large organizations to a greater or lesser extent even today): large volumes of raw transaction data fed into programs that update a master file, with fixed-format reports written to paper.

The term **Data Management Systems** refers to an expansion of this concept, where the raw data, previously copied manually from paper to punched cards, and later into data-entry terminals, is now fed into the system from a variety of sources, including ATMs, EFT, and direct customer entry through the Internet. The master file concept has been largely displaced by database management systems, and static reporting replaced or augmented by ad-hoc reporting and direct inquiry, including downloading of data by customers. The ubiquity of the Internet and the Personal Computer have been the driving force in the transformation of Data Processing to the more global concept of Data Management Systems.

1.3 File Oriented Approach

The earliest business computer systems were used to process business records and produce information. They were generally faster and more accurate than equivalent manual systems. These systems stored groups of records in separate files, and so they were called **file processing systems.** In a typical file processing systems, each department has its own files, designed specifically for those applications. The department itself working with the data processing staff, sets policies or standards for the format and maintenance of its files.

Programs are dependent on the files and vice-versa; that is, when the physical format of the file is changed, the program has also to be changed. Although the traditional file

oriented approach to information processing is still widely used, it does have some very important disadvantages.

1.4 Database Oriented Approach to Data Management

Consider part of a savings-bank enterprise that keeps information about all customers and savings accounts. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the system has a number of application programs that manipulate the files, including A program to debit or credit an account

A program to add a new account

A program to find the balance of an account

A program to generate monthly statements

System programmers wrote these application programs to meet the needs of the bank. New application programs are added to the system as the need arises. For example, suppose that the savings bank decides to offer checking accounts. As a result, the bank creates new permanent files that contain information about all the checking accounts maintained in the bank, and it may have to write new application programs to deal with situations that do not arise in savings accounts, such as overdrafts. Thus, as time goes by, the system acquires more files and more application programs.

This typical **file-processing system** is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) came along, organizations usually stored information in such systems.

Keeping organizational information in a file-processing system has a number of major disadvantages:

Data redundancy and inconsistency.

Since different programmers create the files and application programs over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, the address and telephone number of a particular customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies

of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.

Difficulty in accessing data.

Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* customers. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same officer needs to trim that list to include only those customers who have an account balance of $10,000 or more. As expected, a program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory.

The point here is that conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. More responsive data-retrieval systems are required for general use.

Data isolation. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

Integrity problems. The data values stored in the database must satisfy certain types of **consistency constraints**. For example, the balance of a bank account may never fall below a prescribed amount (say, $25). Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

Atomicity problems. A computer system, like any other mechanical or electrical device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $50 from account *A* to account *B*. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account *A* but was not credited to account *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is

difficult to ensure atomicity in a conventional file-processing system. **Concurrent-access anomalies**. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data. Consider bank account *A*, containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account *A* at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $500, and write back $450 and $400, respectively. Depending on which one writes the value last, the account may contain either $450 or $400, rather than the correct value of $350. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

Security problems. Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult.

These difficulties, among others, prompted the development of database systems. In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems. In most of this book, we use a bank enterprise as a running example of a typical data-processing application found in a corporation.

1.5 Characteristics of Database

The database approach has some very characteristic features which are discussed in detail below:

1.5.1 Concurrent Use

A database system allows several users to access the database concurrently. Answering different questions from different users with the same (base) data is a central aspect of an information system. Such concurrent use of data increases the economy of a system.

An example for concurrent use is the travel database of a bigger travel agency. The employees of different branches can access the database concurrently and book journeys for their clients. Each travel agent sees on his interface if there are still seats available for a specific journey or if it is already fully booked.

1.5.2 Structured and Described Data

A fundamental feature of the database approach is that the database system does not only contain the data but also the complete definition and description of these data. These descriptions are basically details about the extent, the structure, the type and the format of all data and, additionally, the relationship between the data. This kind of stored data is called metadata ("data about data").

1.5.3 Separation of Data and Applications

As described in the feature structured data the structure of a database is described through *metadata* which is also stored in the database. An application software does not need any knowledge about the physical data storage like encoding, format, storage place, etc. It only communicates with the management system f a database (DBMS) via a standardized interface with the help of a standardized language like SQL. The access to the data and the metadata is entirely done by the DBMS. In this way all the applications can be totally separated from the data. Therefore database internal reorganizations or improvement of efficiency do not have any influence on the application software.

1.5.4 Data Integrity

Data integrity is a byword for the quality and the reliability of the data of a database system. In a broader sense data integrity includes also the protection of the database from unauthorized access (confidentiality) and unauthorized changes. Data reflect facts of the real world database.

1.5.5 Transactions

A transaction is a bundle of actions which are done within a database to bring it from one

consistent state to a new consistent state. In between the data are inevitable inconsistent. A transaction is atomic what means that it cannot be divided up any further. Within a transaction all or none of the actions need to be carried out. Doing only a part of the actions would lead to an inconsistent database state. One example of a transaction is the transfer of an amount of money from one bank account to another. The debit of the money from one account and the credit of it to another account makes together a consistent transaction. This transaction is also atomic. The debit or credit alone would both lead to an inconsistent state. After finishing the transaction (debit and credit) the changes to both accounts become persistent and the one who gave the money has now less money on his account while the receiver has now a higher balance.

1.5.6 Data Persistence

Data persistence means that in a DBMS all data is maintained as long as it is not deleted explicitly. The life span of data needs to be determined directly or indirectly be the user and must not be dependent on system features. Additionally data once stored in a database must not be lost. Changes of a database which are done by a transaction are persistent. When a transaction is finished even a system crash cannot put the data in danger.

1.6 Advantages and Disadvantages of a DBMS

Using a DBMS to manage data has many advantages:

Data independence: Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.

Efficient data access: A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.

Data integrity and security: If data is always accessed through the DBMS, the DBMS can enforce integrity constraints on the data. For example, before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded. Also, the DBMS can enforce *access controls* that govern what data is visible to different classes of users.

Data administration: When several users share the data, centralizing the administration of data can offer significant improvements. Experienced professionals, who understand the nature of the data being managed, and how different groups of users use it, can be responsible for organizing the data representation to minimize redundancy and fine-tuning the storage of the data to make retrieval efficient.

Concurrent access and crash recovery: A DBMS schedules concurrent accesses to the data in such a manner that users can think of the data as being accessed by only one user at a time. Further, the DBMS protects users from the effects of system failures.

Reduced application development time: Clearly, the DBMS supports many important functions that are common to many applications accessing data stored in the DBMS. This, in conjunction with the high-level interface to the data, facilitates quick development of applications. Such applications are also likely to be more robust than applications developed from scratch because many important tasks are handled by the DBMS instead of being implemented by the application. Given all these advantages, is there ever a reason *not* to use a DBMS? A DBMS is a complex piece of software, optimized for certain kinds of workloads (e.g., answering complex queries or handling many concurrent requests), and its performance may not be adequate for certain specialized applications. Examples include applications with tight real-time constraints or applications with just a few well-designed critical operations for which efficient custom code must be written. Another reason for not using a DBMS is that an application may need to manipulate the data in ways not supported by the query language. In such a situation, the abstract view of the data presented by the DBMS does not match the application's needs, and actually gets in the way. As an example, relational databases do not support flexible analysis of text data (although vendors are now extending their products in this direction). If specialized performance or data manipulation requirements are central to an application, the application may choose not to use a DBMS, especially if the added benefits of a DBMS (e.g., flexible querying, security, concurrent access, and crash recovery) are not required. In most situations calling for large-scale data management, however, DBMSs have become an indispensable tool.

Disadvantages of a DBMS

Danger of a Overkill: For small and simple applications for single users a database system is often not advisable.

Complexity: A database system creates additional complexity and requirements. The supply and operation of a database management system with several users and databases is quite costly and demanding.

Qualified Personnel: The professional operation of a database system requires appropriately trained staff. Without a qualified database administrator nothing will work for long.

Costs: Through the use of a database system new costs are generated for the system itself but also for additional hardware and the more complex handling of the system. **Lower**

Efficiency: A database system is a multi-use software which is often less efficient than specialized software which is produced and optimized exactly for one problem.

1.7 Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. Schemas are changed infrequently, if at all.

The concept of database schemas and instances can be understood by analogy to a program written in a programming language. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Each variable has a particular value at a given instant. The values of the variables in a program at a point in time correspond to an *instance* of a database schema.

Database systems have several schemas, partitioned according to the levels of abstraction.

The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschemas** that describe different views of the database.

Of these, the logical schema is by far the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema. The physical schema is hidden beneath the logical schema, and can usually be changed easily without affecting application programs. Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

We study languages for describing schemas, after introducing the notion of data models in the next section.

1.8 Data Models

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

To illustrate the concept of a data model, we outline two data models in this section: the entity-relationship model and the relational model. Both provide a way to describe the design of a database at the logical level.

1.8.1 The Entity-Relationship Model

The entity-relationship (E-R) data model is based on a perception of a real world that consists of a collection of basic objects, called *entities*, and of *relationships* among these objects. An entity is a “thing” or “object” in the real world that is distinguishable from

other objects. For example, each person is an entity, and bank accounts can be considered as entities.

Entities are described in a database by a set of **attributes**. For example, the attributes *account-number* and *balance* may describe one particular account in a bank, and they form attributes of the *account* entity set. Similarly, attributes *customer-name*, *customer-street* address and *customer-city* may describe a *customer* entity.

An extra attribute *customer-id* is used to uniquely identify customers (since it may be possible to have two customers with the same name, street address, and city).

A unique customer identifier must be assigned to each customer. In the United States, many enterprises use the social-security number of a person (a unique number the U.S. government assigns to every person in the United States) as a customer identifier.

A **relationship** is an association among several entities. For example, a *depositor* relationship associates a customer with each account that she has. The set of all entities of the same type and the set of all relationships of the same type are termed an **entity set** and **relationship set**, respectively.

The overall logical structure (schema) of a database can be expressed graphically by an *E-R diagram.*

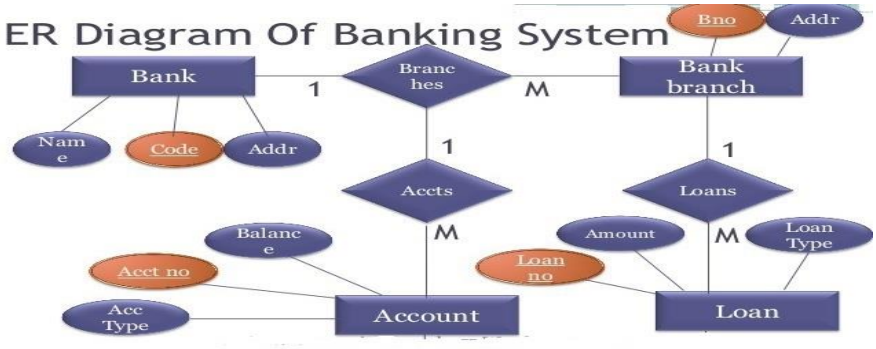


Figure 1.1 ER Diagram of Banking System

1.8.2 Relational Model

The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name.

The data is arranged in a relation which is visually represented in a two dimensional table. The data is inserted into the table in the form of tuples (which are nothing but rows). A tuple is formed by one or more than one attributes, which are used as basic building blocks in the formation of various expressions that are used to derive a meaningful information. There can be any number of tuples in the table, but all the tuple contain fixed and same attributes with varying values. The relational model is

implemented in database where a relation is represented by a table, a tuple is represented by a row, an attribute is represented by a column of the table, attribute name is the name of the column such as 'identifier', 'name', 'city' etc., attribute value contains the value for column in the row. Constraints are applied to the table and form the logical schema. In order to facilitate the selection of a particular row/tuple from the table, the attributes i.e. column names are used, and to expedite the selection of the rows some fields are defined uniquely to use them as indexes, this helps in searching the required data as fast as possible. All the relational algebra operations, such as Select, Intersection, Product, Union, Difference, Project, Join, Division, Merge etc. can also be performed on the Relational Database Model. Operations on the Relational Database Model are facilitated with the help of different conditional expressions, various key attributes, pre-defined constraints etc.

Columns of table representing attributes

Primary key attribute

Cell representing value, Mistry of CAND_LNAME of candidate 23456

Candidate Table

CAND_ID	CAND_LNAME	CAND_FNAME	CAND_MI	CAND_DOB	CAND_QUA	CAND_MAJOR	CAND_EXP
12345	Panchal	Champak	D.	6/12/1985	B.Sc.	IT	1 year
23456	Mistry	Kunal	J.	8/19/1988	BCA	IT	0
34567	Rawal	Sai	N.	8/21/1975	B.E..	Comp. Sc.	8 years
45678	Shah	Usha	R.	2/6/1980	B.E.	Comp. Sc.	6 years
56789	Patel	Manisha	L.	9/9/1981	MCA	IT	6 years
67890	Patel	Abha	P.	8/8/1985	M.Sc.	CA & IT	3 years
78901	Joiner	Misty	E.	5/2/1980	M.S.	Comp. Sc.	7 years
89012	Shah	Kirti	K.	5/10/1984	M.Sc.	CIS	2 years
90123	Judd	Ashley	W.	3/3/1992	B.S.	CIS	3 years
13456	Jones	Frank	P.	1/8/1980	B.S.	IT	6 years

Figure 1.2 Candidate relation

1.8.3 Other Data Models

The **object-oriented data model** is another data model that has seen increasing attention. The object-oriented model can be seen as extending the E-R model with notions object-oriented data model.

The **object-relational data model** combines features of the object-oriented data model and relational data model. Semi structured data models permit the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast with the data models mentioned earlier, where every data item of a particular type must have the same set of attributes. The **extensible markup language (XML)** is widely used to represent semi structured data.

Historically, two other data models, the **network data model** and the **hierarchical data model**, preceded the relational data model. These models were tied closely to the underlying implementation, and complicated the task of modeling data. As a result they

are little used now, except in old database code that is still in service in some places. They are outlined in Appendices A and B, for interested readers.

1.9 Database Languages

A database system provides a **data definition language** to specify the database schema and a **data manipulation language** to express database queries and updates. In practice, the data definition and data manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

1.9.1 Data-Definition Language

We specify a database schema by a set of definitions expressed by a special language called a **data-definition language** (**DDL**).

For instance, the following statement in the SQL language defines the *account* table:

create table *account (account-number* ***char****(10), balance* ***integer****)*

Execution of the above DDL statement creates the *account* table. In addition, it updates a special set of tables called the **data dictionary** or **data directory**.

A data dictionary contains **metadata**—that is, data about data. The schema of a table is an example of metadata. A database system consults the data dictionary before reading or modifying actual data.

We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a **data storage and definition** language. These statements define the implementation details of the database schemas, which are usually hidden from the users.

The data values stored in the database must satisfy certain **consistency constraints**. For example, suppose the balance on an account should not fall below $100. The DDL provides facilities to specify such constraints. The database systems check these constraints every time the database is updated.

1.9.2 Data-Manipulation Language

Data manipulation is

i) The retrieval of information stored in the database
ii) The insertion of new information into the database
iii) The deletion of information from the database
iv) The modification of information stored in the database

A **data-manipulation language (DML)** is a language that enables users to access or manipulate data as organized by the appropriate data model. There are basically two types:

Procedural DMLs require a user to specify *what* data are needed and *how* to get those data.

Declarative DMLs (also referred to as **nonprocedural** DMLs) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs. However, since a user does not have to specify how to get the data, the database system has to figure out an efficient means of accessing data. The DML component of the SQL language is nonprocedural.

A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**. Although technically incorrect, it is common practice to use the terms *query language* and *data manipulation language* synonymously.

This query in the SQL language finds the name of the customer whose customer-id is 192-83-7465:

select *customer.customer-name*
from *customer*
where *customer.customer-id* = 192-83-7465

The query specifies that those rows *from* the table *customer where* the *customer-id* is 192-83-7465 must be retrieved, and the *customer-name* attribute of these rows must be displayed.

Queries may involve information from more than one table. For instance, the following query finds the balance of all accounts owned by the customer with customerid 192-83-7465.

select *account.balance*
from *depositor, account*
where *depositor.customer-id* = 192-83-7465 **and**
depositor.account-number = account.account-number

There are a number of database query languages in use, either commercially or experimentally.

The levels of abstraction apply not only to defining or structuring data, but also to manipulating data. At the physical level, we must define algorithms that allow efficient access to data. At higher levels of abstraction, we emphasize ease of use. The goal is to allow humans to interact efficiently with the system. The query processor component of the database system translates DML queries into sequences of actions at the physical level of the database system.

1.10 Data Dictionary

We can define a data dictionary as a DBMS component that stores the definition of data characteristics and relationships. You may recall that such "data about data" were labeled metadata. The DBMS data dictionary provides the DBMS with its self describing characteristic. In effect, the data dictionary resembles and X-ray of the company's entire data set, and is a crucial element in the data administration function.

The two main types of data dictionary exist, integrated and stand alone. An integrated data dictionary is included with the DBMS. For example, all relational DBMSs include a built in data dictionary or system catalog that is frequently accessed and updated by the RDBMS. Other DBMSs especially older types, do not have a built in data dictionary instead the DBA may use third party stand alone data dictionary systems.

Data dictionaries can also be classified as active or passive. An active data dictionary is automatically updated by the DBMS with every database access, thereby keeping its access information up-to-date. A passive data dictionary is not updated automatically and usually requires a batch process to be run. Data dictionary access information is normally used by the DBMS for query optimization purpose.

The data dictionary's main function is to store the description of all objects that interact with the database. Integrated data dictionaries tend to limit their metadata to the data managed by the DBMS. Stand alone data dictionary systems are more usually more flexible and allow the DBA to describe and manage all the organization's data, whether or not they are computerized. Whatever the data dictionary's format, its existence provides database designers and end users with a much improved ability to communicate. In addition, the data dictionary is the tool that helps the DBA to resolve data conflicts.

Although, there is no standard format for the information stored in the data dictionary several features are common. For example, the data dictionary typically stores descriptions of all:

- Data elements that are define in all tables of all databases. Specifically the data dictionary stores the name, datatypes, display formats, internal storage formats, and validation rules. The data dictionary tells where an element is used, by whom it is used and so on.
- Tables define in all databases. For example, the data dictionary is likely to store the name of the table creator, the date of creation access authorizations, the number of columns, and so on.

- Indexes define for each database tables. For each index the DBMS stores at least the index name the attributes used, the location, specific index characteristics and the creation date.
- Define databases: who created each database, the date of creation where the database is located, who the DBA is and so on.
- End users and The Administrators of the data base
- Programs that access the database including screen formats, report formats application formats, SQL queries and so on.
- Access authorization for all users of all databases.
- Relationships among data elements which elements are involved: whether the relationship is mandatory or optional, the connectivity and cardinality and so on.

If the data dictionary can be organized to include data external to the DBMS itself, it becomes an especially flexible to for more general corporate resource management. The management of such an extensive data dictionary, thus, makes it possible to manage the use and allocation of all of the organization information regardless whether it has its roots in the database data. This is why some managers consider the data dictionary to be the key element of the information resource management function. And this is also why the data dictionary might be described as the information resource dictionary.

The metadata stored in the data dictionary is often the bases for monitoring the database use and assignment of access rights to the database users. The information stored in the database is usually based on the relational table format, thus, enabling the DBA to query the database with SQL command. For example, SQL command can be used to extract information about the users of the specific table or about the access rights of a particular user.

1.11 Database Administrators and Database Users

A primary goal of a database system is to retrieve information from and store new information in the database. People who work with a database can be categorized as database users or database administrators.

1.11.1 Database Users and User Interfaces

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account *A* to account *B* invokes a program called *transfer*.

This program asks the teller for the amount of money to be transferred, the account from which the money is to be transferred, and the account to which the money is to be transferred.

As another example, consider a user who wishes to find her account balance over the World Wide Web. Such a user may access a form, where she enters her account number. An application program at the Web server then retrieves the account balance, using the given account number, and passes this information back to the user. The typical user interface for naive users is a forms interface, where the user can fill in appropriate fields of the form. Naive users may also simply read *reports* generated from the database.

Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces.

Rapid application development (RAD) tools are tools that enable an application programmer to construct forms and reports without writing a program. There are also special types of programming languages that combine imperative control structures (for example, for loops, while loops and if-then-else statements) with statements of the data manipulation language. These languages, sometimes called *fourth-generation languages*, often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth generation language.

Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a **query processor**, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category.

Online analytical processing (OLAP) tools simplify analysts' tasks by letting them view summaries of data in different ways. For instance, an analyst can see total sales by region (for example, North, South, East, and West), or by product, or by a combination of region and product (that is, total sales of each product in each region). The tools also permit the analyst to select specific regions, look at data in more detail (for example, sales by city within a region) or look at the data in less detail (for example, aggregate products together by category).

Another class of tools for analysts is **data mining** tools, which help them find certain kinds of patterns in data.

Specialized users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework.

Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

1.11.2 Database Administrator

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator** (**DBA**). The functions of a DBA include:

i) Schema definition. The DBA creates the original database schema by executing a set of data definition statements in the DDL.

Storage structure and access-method definition.

ii) Schema and physical-organization modification. The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.

iii) Granting of authorization for data access. By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.

iv) Routine maintenance. Examples of the database administrator's routine maintenance activities are:

Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.

Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.

Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

1.12 DBMS Architecture and Data Independence

Three important characteristics of the database approach are (1) insulation of programs and data (program-data and program-operation independence); (2) support of multiple user views; and (3) use of a catalog to store the database description (schema). In this section we specify an architecture for database systems, called the **three-schema architecture**, which was proposed to help achieve and visualize these characteristics. We then discuss the concept of data independence.

1.12.1 The Three-Schema Architecture

The goal of the three-schema architecture, illustrated in Figure 1.1, is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

1. The **internal level** has an **internal schema,** which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The **conceptual level** has a **conceptual schema,** which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
3. The **external** or **view level** includes a number of **external schemas** or **user views.** Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

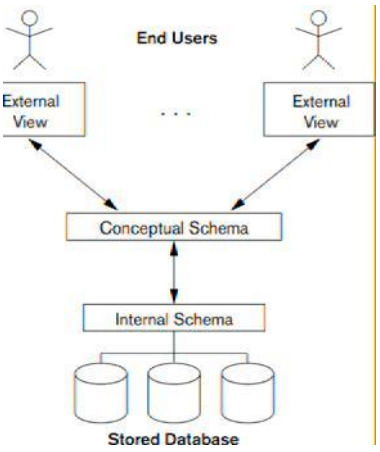


Figure 1.3 The Three Schema Architecture

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely, but support the three-schema architecture to some extent. Some DBMSs may include physical-level details in the conceptual schema. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.

Notice that the three schemas are only *descriptions* of data; the only data that *actually* exists is at the physical level. In a DBMS based on the three-schema architecture, each

user group refers only to its own external schema. Hence, the DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The processes of transforming requests and results between levels are called **mappings.** These mappings may be time-consuming, so some DBMSs—especially those that are meant to support small databases—do not support external views. Even in such systems, however, a certain amount of mapping is necessary to transform requests between the conceptual and internal levels.

1.12.2 Data Independence

The three-schema architecture can be used to explain the concept of **data independence,** which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. **Logical data independence** is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected. Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.
2. **Physical data independence** is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized—for example, by creating additional access structures—to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema.

Whenever we have a multiple-level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. The DBMS uses additional software to accomplish these mappings by referring to the mapping information in the catalog. Data independence is accomplished because, when the schema

is changed at some level, the schema at the next higher level remains unchanged; only the *mapping* between the two levels is changed. Hence, application programs referring to the higher-level schema need not be changed.

The three-schema architecture can make it easier to achieve true data independence, both physical and logical. However, the two levels of mappings create an overhead during compilation or execution of a query or program, leading to inefficiencies in the DBMS. Because of this, few DBMSs have implemented the full three-schema architecture.

1.13 Types of Database System

Several criteria are normally used to classify DBMSs. The *first* is the data model on which the DBMS is based. The main data model used in many current commercial DBMSs is the relational data model. The object data model was implemented in some commercial systems but has not had widespread use. Many legacy (older) applications still run on database systems based on the hierarchical and network data models. The relational DBMSs are evolving continuously, and, in particular, have been incorporating many of the concepts that were developed in object databases. This has led to a new class of DBMSs called object-relational DBMSs. We can hence categorize DBMSs based on the *data model*: **relational, object, object-relational, hierarchical, network, and other**. The *second* criterion used to classify DBMSs is the number of users supported by the system. **Single-user systems** support only one user at a time and are mostly used with personal computers. **Multiuser systems**, which include the majority of DBMSs, support multiple users concurrently. A *third* criterion is the number of sites over which the database is distributed. A DBMS is centralized if the data is stored at a single computer site. A **centralized DBMS** can support multiple users, but the DBMS and the database themselves reside totally at a single computer site. A **distributed DBMS** (DDBMS) can have the actual database and DBMS software distributed over many sites, connected by a computer network. Homogeneous DDBMSs use the same DBMS software at multiple sites. A recent trend is to develop software to access several autonomous preexisting databases stored under heterogeneous DDBMSs. This leads to a federated DBMS (or multi-database system), in which the participating DBMSs are loosely coupled and have a degree of local autonomy. Many DBMSs use a client-server architecture.

1.14Tier Architecture:

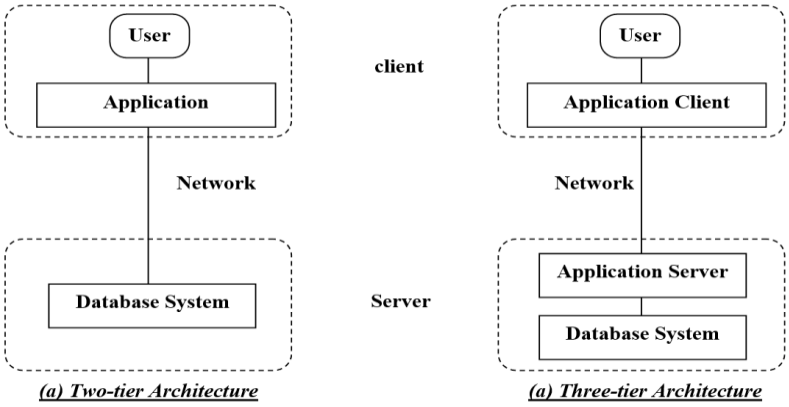


Figure 1.4 DBMS Tier Architecture

The design of a DBMS depends on its architecture. It can be centralized or decentralized or hierarchical. The architecture of a DBMS can be seen as either single tier or multi-tier. An n-tier architecture divides the whole system into related but independent n modules, which can be independently modified, altered, changed, or replaced. In 1-tier architecture, the DBMS is the only entity where the user directly sits on the DBMS and uses it. Any changes done here will directly be done on the DBMS itself. It does not provide handy tools for end-users. Database designers and programmers normally prefer to use single-tier architecture. If the architecture of DBMS is 2-tier, then it must have an application through which the DBMS can be accessed. Programmers use 2-tier architecture where they access the DBMS by means of an application. Here the application tier is entirely independent of the database in terms of operation, design, and programming.

A 3-tier architecture separates its tiers from each other based on the complexity of the users and how they use the data present in the database. It is the most widely used architecture to design a DBMS.

Database (Data) Tier: At this tier, the database resides along with its query processing languages. We also have the relations that define the data and their constraints at this level.

Application (Middle) Tier: At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. End-users are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the end-user and the database.

User (Presentation) Tier: End-users operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer, multiple views of the database can be provided by the application. All views are generated by applications that reside in the application tier. Multiple-tier database architecture is highly modifiable, as almost all its components are independent and can be changed independently.

We summarize the discussion below:

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. This is a collection of related data with an implicit meaning and hence is a database.

A **datum** – a unit of data – is a symbol or a set of symbols which is used to represent something. This relationship between symbols and what they represent is the essence of what we mean by **information**.

Knowledge refers to the practical use of information.

The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**.

The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at the logical level. A database may also have several schemas at the view level, sometimes called **subschema**, that describe different views of the database.

Application programs are said to exhibit **physical data independence** if they do not depend on the physical schema, and thus need not be rewritten if the physical schema changes.

Underlying the structure of a database is the **data model**: a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

A database system provides a **data definition language** to specify the database schema and a **data manipulation language** to express database queries and updates.

One of the main reasons for using DBMSs is to have central control of both the data and the programs that access those data. A person who has such central control over the system is called a **database administrator** (**DBA**).

MODULE 1 MCQ and Short type Problem

Multiple choice questions (MCQ)

1. _____refers to the basic facts and entities, such as names and numbers.

a. Data
b. Information
c. Input
d. Output

Ans. (a)

2. MIS stands for:

a. Management Information Server
b. Management Information Service

c. Management Information System
d. Master Information System
Ans. (c)

3. Which is the data model
a. Relational
b. Object-Relational
c. Network
d. All of these
Ans. (a)

4. Which is not the feature of database:
a. Data redundancy
b. Independence
c. Flexibility
d. Data Integrity
Ans. (a)

5. Which is the type of data independence:
a. Physical data independence
b. Logical data independence
c. Both
d. None of these
Ans. (c)

6. Which is the advantage of database:
a. Prevents Data redundancy
b. Restricts unauthorized access
c. Persistent storage
d. All of these
Ans. (d)

7. Which is the database language:
a. C
b. C++
c. SQL
d. None of these
Ans. (c)

8. Which person is responsible for overall activities for database:
a. Database designer
b. Database analyst
c. Database Administrator
d. Database manager
Ans. (c)

9. Which level of database is viewed by user:
a. Internal level
b. External Level
c. Conceptual Level
d. All of these
Ans. (b)

10. Internal level has:

a. Individual Users View of the database
b. Community view of the database
c. Physical Representation of the database
d. All of these
Ans. (c)

11. Which is the component of database management system:
a. Query Language
b. Database Manager
c. File manager
d. All of these
Ans. (d)

12. _____is the structure of the database.
a. Table
b. Relation
c. Schema
d. None of these
Ans. (c)

13. Schema is usually stored in_____.
a. Tables
b. Data Dictionary
c. Both
d. None of these
Ans. (c)

14. Schema is defined by:
a. DML
b. DDL
c. DCL
d. DQL
Ans. (b)

15. DML language is used to:
a. Define schema
b. Define internal level
c. Access Data
d. All of these
Ans. (d)

16. DBMS is the bridge between operating system and _____.
a. User
b. Database administrator
c. Application program
d. None of these
Ans. (c)

17. Which is the most popular database model:
a. Network Model
b. Relational Model
c. Hierarchical Model
d. Object Oriented
Ans. (b)

Short question

1. Why would you choose a database system instead of simply storing data in operating system files?
2. Explain the difference between logical and physical data independence.
3. What is DDL, DML and DCL?

Long type question:

1. What is data dictionary? Explain Tier architecture?
2. Explain 3 schema architecture and data abstraction.
3. Discuss functions of DBA, Database user. Define instance, schema.

Text Books:

1. Henry F. Korth and Silberschatz Abraham, "Database System Concepts", Mc.Graw Hill.
2. Elmasri Ramez and Novathe Shamkant, "Fundamentals of Database Systems", Benjamin Cummings Publishing. Company.
3. Ramakrishnan: Database Management System , McGraw-Hill

Module 2:

Entity-Relationship and Relational Database Model [11L]

Basic concepts, Design Issues, Mapping Constraints, Keys, Entity-Relationship Diagram, Weak Entity Sets, Extended E-R features, case study on E-R Model. Structure of relational Databases, Relational Algebra, Relational Calculus, Extended Relational Algebra Operations, Views, Modifications of the Database.

2.1 Basic Concepts:

A data model is a conceptual representation of the data structures that are required by a database. The data structures include the data objects, the associations between data objects, and the rules which govern operations on the objects. As the name implies, the data model focuses on what data is required and how it should be organized rather than what operations will be performed on the data. To use a common analogy, the data model is equivalent to an architect's building plans.

A data model is independent of hardware or software constraints. Rather than try to represent the data as a database would see it, the data model focuses on representing the data as the user sees it in the "real world". It serves as a bridge between the concepts that make up real-world events and processes and the physical representation of those concepts in a database.

Methodology

There are two major methodologies used to create a data model: the Entity-Relationship (ER) approach and the Object Model. This document uses the Entity-Relationship approach.

2.2 Design Issues:

Data Modeling In the Context of Database Design

Database design is defined as: "design the logical and physical structure of one or more databases to accommodate the information needs of the users in an organization for a defined set of applications". The design process roughly follows five steps:

1. Planning and analysis
2. Conceptual design
3. Logical design
4. Physical design
5. Implementation

The data model is one part of the conceptual design process. The other, typically is the functional model. The data model focuses on what data should be stored in the database while the functional model deals with how the data is processed. To put this in the context of the relational database, the data model is used to design the relational tables. The functional model is used to design the queries which will access and perform operations on those tables.

Components of a Data Model

The data model gets its inputs from the planning and analysis stage. Here the modeler, along with analysts, collects information about the requirements of the database by reviewing existing documentation and interviewing end-users.

The data model has two outputs. The first is an entity-relationship diagram which represents the data structures in a pictorial form. Because the diagram is easily learned, it is valuable tool to communicate the model to the end-user. The second component is a data document. This is a document that describes in detail the data objects, relationships, and rules required by the database. The dictionary provides the detail required by the database developer to construct the physical database.

Why is Data Modeling Important?

Data modeling is probably the most labor intensive and time consuming part of the development process. Why bother especially if you are pressed for time? A common response by practitioners who write on the subject is that you should no more build a database without a model than you should build a house without blueprints.

The goal of the data model is to make sure that the all data objects required by the database are completely and accurately represented. Because the data model uses easily understood notations and natural language, it can be reviewed and verified as correct by the end-users.

The data model is also detailed enough to be used by the database developers to use as a "blueprint" for building the physical database. The information contained in the data model will be used to define the relational tables, primary and foreign keys, stored procedures, and triggers. A poorly designed database will require more time in the long-term. Without careful planning you may create a database that omits data required to create critical reports,

produces results that are incorrect or inconsistent, and is unable to accommodate changes in the user's requirements.

2.3 The Entity-Relationship Model

The Entity-Relationship (ER) model was originally proposed by Peter in 1976 as a way to unify the network and relational database views. Simply stated the ER model is a conceptual data model that views the real world as entities and relationships. A basic component of the model is the Entity-Relationship diagram which is used to visually represent data objects. Since Chen wrote his paper the model has been extended and today it is commonly used for database design for the database designer, the utility of the ER model is:

It maps well to the relational model. The constructs used in the ER model can easily be transformed into relational tables.

It is simple and easy to understand with a minimum of training. Therefore, the model can be used by the database designer to communicate the design to the end user.

In addition, the model can be used as a design plan by the database developer to implement a data model in a specific database management software.

Basic Constructs of E-R Modeling

The ER model views the real world as a construct of entities and association between entities.

Entities

Entities are the principal data object about which information is to be collected. Entities are usually recognizable concepts, either concrete or abstract, such as person, places, things, or events which have relevance to the database. Some specific examples of entities are EMPLOYEES, PROJECTS, INVOICES. An entity is analogous to a table in the relational model.

Entities are classified as independent or dependent (in some methodologies, the terms used are strong and weak, respectively). An independent entity is one that does not rely on another for identification. A dependent entity is one that relies on another for identification.

An entity occurrence (also called an instance) is an individual occurrence of an entity. An occurrence is analogous to a row in the relational table.

Special Entity Types

Associative entities (also known as intersection entities) are entities used to associate two or more entities in order to reconcile a many-to-many relationship.

Subtypes entities are used in generalization hierarchies to represent a subset of instances of their parent entity, called the super type, but which have attributes or relationships that apply only to the subset.

Associative entities and generalization hierarchies are discussed in more detail below.

Relationships

A Relationship represents an association between two or more entities. An example of a relationship would be:

Employees are assigned to projects

Projects have subtasks

Departments manage one or more projects

Relationships are classified in terms of degree, connectivity, cardinality, and existence.

Attributes

Attributes describe the entity of which they are associated. A particular instance of an attribute is a value. For example, "Jane R. Hathaway" is one value of the attribute Name. The domain of an attribute is the collection of all possible values an attribute can have. The domain of Name is a character string.

Attributes can be classified as identifiers or descriptors. Identifiers, more commonly called keys, uniquely identify an instance of an entity. A descriptor describes a non-unique characteristic of an entity instance.

Classifying Relationships

Relationships are classified by their degree, connectivity, cardinality, direction, type, and existence. Not all modeling methodologies use all these classifications.

Degree of a Relationship

The degree of a relationship is the number of entities associated with the relationship. The n-ary relationship is the general form for degree n. Special cases are the binary and ternary where the degree is 2, and 3, respectively.

Binary relationships: the association between two entities is the most common type in the real world. A recursive binary relationship occurs when an entity is related to itself. An example might be "some employees are married to other employees".

A ternary relationship involves three entities and is used when a binary relationship is inadequate. Many modeling approaches recognize only binary relationships. Ternary or n-ary relationships are decomposed into two or more binary relationships.

2.4 Mapping Constraints

Connectivity and Cardinality The connectivity of a relationship describes the mapping of associated entity instances in the relationship. The values of connectivity are "one" or "many". The cardinality of a relationship is the actual number of related occurrences for each of the two entities. The basic types of connectivity for relations are: one-to-one, one-to-many, and many-to-many.

A *one-to-one* (1:1) relationship is when at most one instance of a entity A is associated with one instance of entity B. For example, "employees in the company are each assigned their own office. For each employee there exists a unique office and for each office there exists a unique employee.

A *one-to-many* (1:N) relationships is when for one instance of entity A, there are zero, one, or many instances of entity B, but for one instance of entity B, there is only one instance of entity A. An example of a 1:N relationships is

A department has many employees

Each employee is assigned to one department

A *many-to-many* (M:N) relationship, sometimes called non-specific, is when for one instance of entity A, there are zero, one, or many instances of entity B and for one instance of entity B there are zero, one, or many instances of entity A. An example is: employees can be assigned to no more than two projects at the same time; projects must have assigned at least three employees

A single employee can be assigned to many projects; conversely, a single project can have assigned to it many employee. Here the cardinality for the relationship between employees and projects is two and the cardinality between project and employee is three. Many-to-many relationships cannot be directly translated to relational tables but instead must be transformed into two or more one-to-many relationships using associative entities.

Direction

The direction of a relationship indicates the originating entity of a binary relationship. The entity from which a relationship originates is the parent entity; the entity where the relationship terminates is the child entity.

The direction of a relationship is determined by its connectivity. In a one-to-one relationship the direction is from the independent entity to a dependent entity. If both entities are independent, the direction is arbitrary. With one-to-many relationships, the entity occurring once is the parent. The direction of many-to-many relationships is arbitrary.

Type

An *identifying relationship* is one in which one of the child entities is also a dependent entity. A *non-identifying relationship* is one in which both entities are independent.

Existence

Existence denotes whether the existence of an entity instance is dependent upon the existence of another, related, entity instance. The existence of an entity in a relationship is defined as either *mandatory* or *optional*. If an instance of an entity must always occur for an entity to be included in a relationship, then it is mandatory. An example of mandatory existence is the

statement "every project must be managed by a single department". If the instance of the entity is not required, it is optional. An example of optional existence is the statement, "employees may be assigned to work on projects".

2.5 Keys

Primary Key – A primary is a column or set of columns in a table that uniquely identifies tuples (rows) in that table.

Super Key – A super key is a set of one of more columns (attributes) to uniquely identify rows in a table.

Candidate Key – A super key with no redundant attribute is known as candidate key.

Alternate Key – Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate or secondary keys.

Composite Key – A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key.

Foreign Key – Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

2.6 Entity Relationship Diagram

Symbol used in E-R diagram

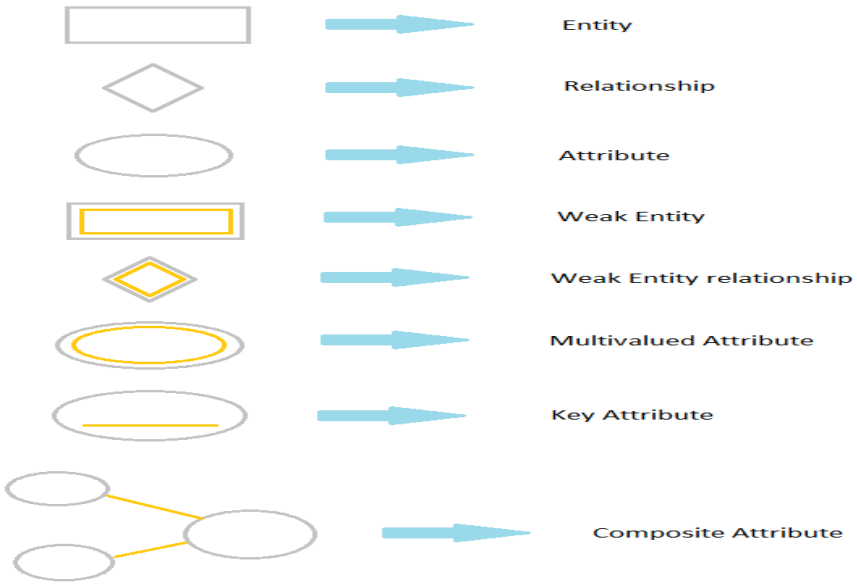


Figure 2.1 Symbols used in ER Diagram

2.7 Weak Entity Set:

- An entity set that does not have a primary key is referred to as a weak entity set.
- An entity set that has primary key is known as strong entity set.
- The existence of a weak entity set depends on the existence of a identifying entity set:
    - It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set.

    - Identifying relationship depicted using a double diamond.
- The discriminator (or partial key) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.
- The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.
- We depict a weak entity set by double rectangles. We underline the discriminator of a weak entity set with a dashed line.
- Example:
    - payment-number – discriminator of the payment entity set
    - Primary key for payment – (loan-number, payment-number).
- Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.
- If loan-number were explicitly stored, payment could be made a strong entity, but then the relationship between payment and loan would be duplicated by an implicit relationship defined by the attribute loan-number common to payment and loan.

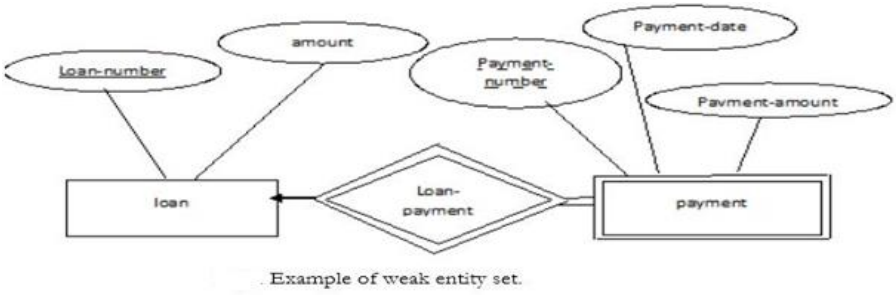


Figure 2.2 Weak Entity set

2.8 Extended E-R features

As the complexity of data increased in the late 1980s, it became more and more difficult to use the traditional ER Model for database modeling. Hence some improvements or enhancements were made to the existing ER Model to make it able to handle the complex applications better.
Hence, as part of the **Enhanced ER Model**, along with other improvements, three new concepts were added to the existing ER Model, they were:

1. Generalization
2. Specialization
3. Aggregation

Let's understand what they are, and why were they added to the existing ER Model.

Generalization

Generalization is a bottom-up approach in which two lower level entities combine to form a higher level entity. In generalization, the higher level entity can also combine with other lower level entities to make further higher level entity.

It's more like Super class and Subclass system, but the only difference is the approach, which is bottom-up. Hence, entities are combined to form a more generalized entity, in other words, sub classes are combined to form a super-class. For example, **Saving** and **Current** account types entities can be generalized and an entity with name **Account** can be created, which covers both.

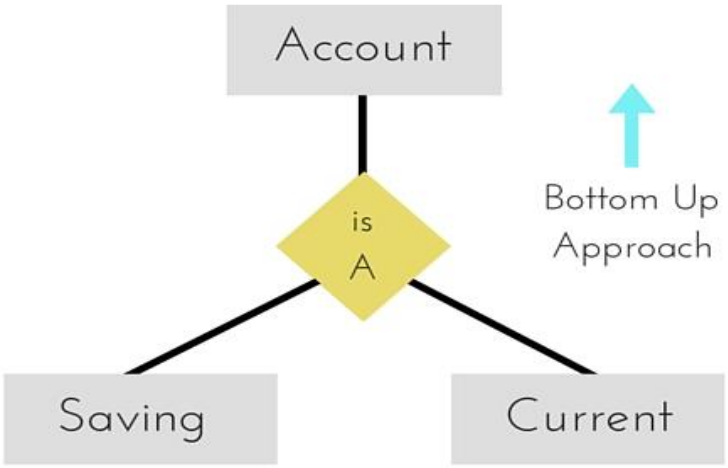


Figure 2.3 Example of Generalization

Specialization

Specialization is opposite to Generalization. It is a top-down approach in which one higher level entity can be broken down into two lower level entity. In specialization, a higher level entity may not have any lower-level entity sets, it's possible.

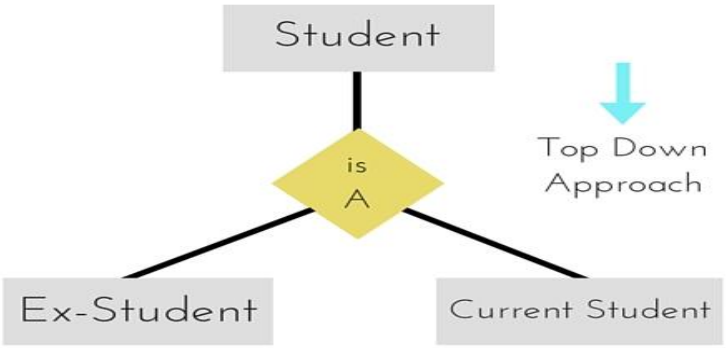


Figure 2.4 Example of Specialization

Aggregation

Aggregation is a process when relation between two entities is treated as a **single entity**.

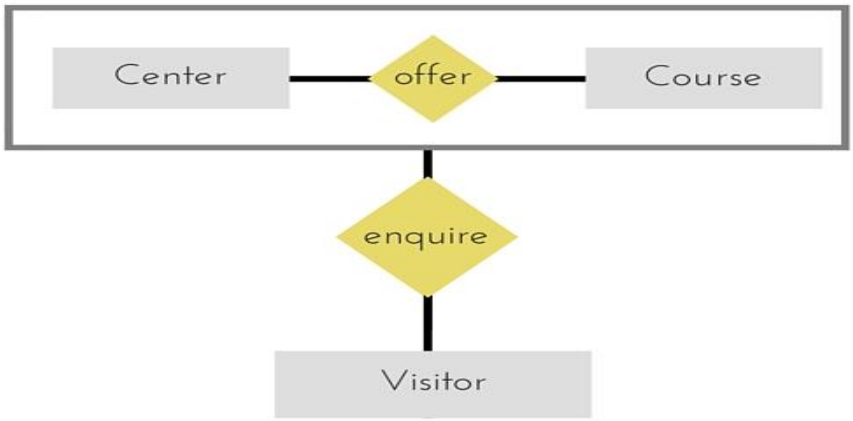


Figure 2.5 Example of Aggregation

In the diagram above, the relationship between **Center** and **Course** together, is acting as an Entity, which is in relationship with another entity **Visitor**. Now in real world, if a Visitor or a Student visits a Coaching Center, he/she will never enquire about the center only or just about the course, rather he/she will ask enquire about both.

2.9 Case study on E-R Model

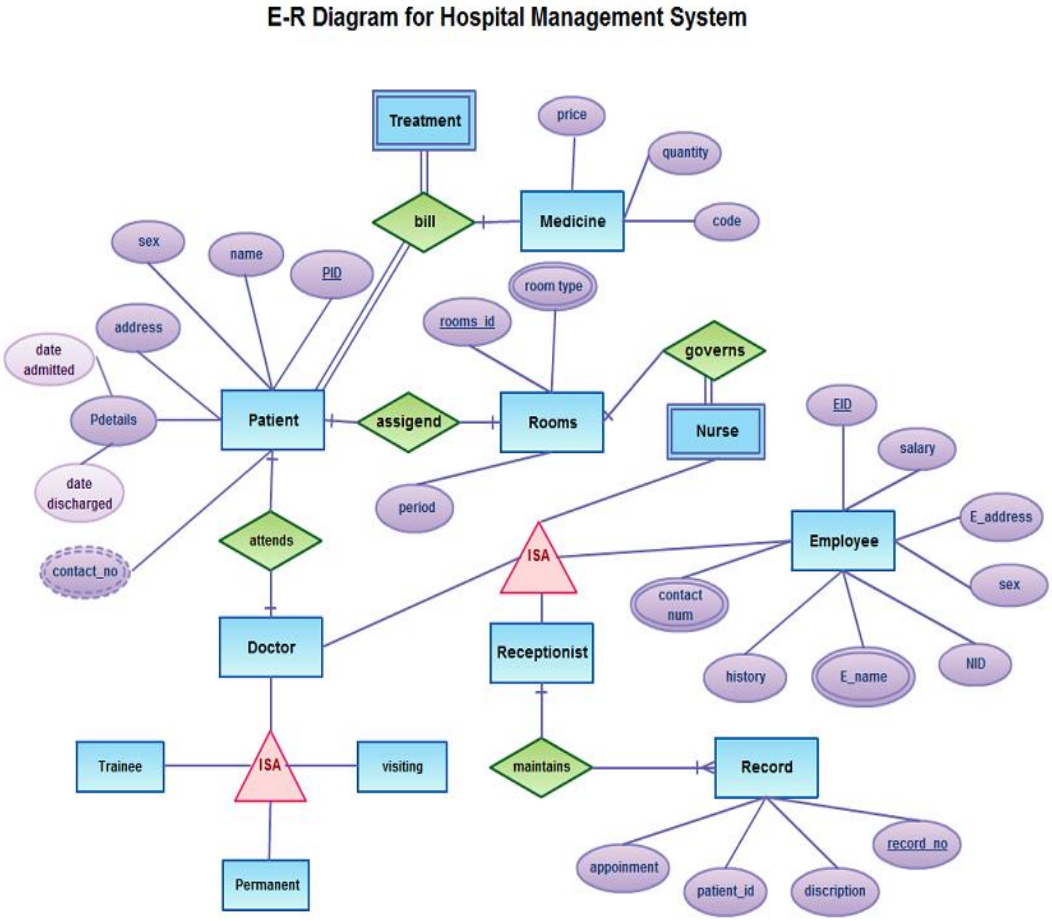


Figure 2.6 E-R diagram of Hospital Management System

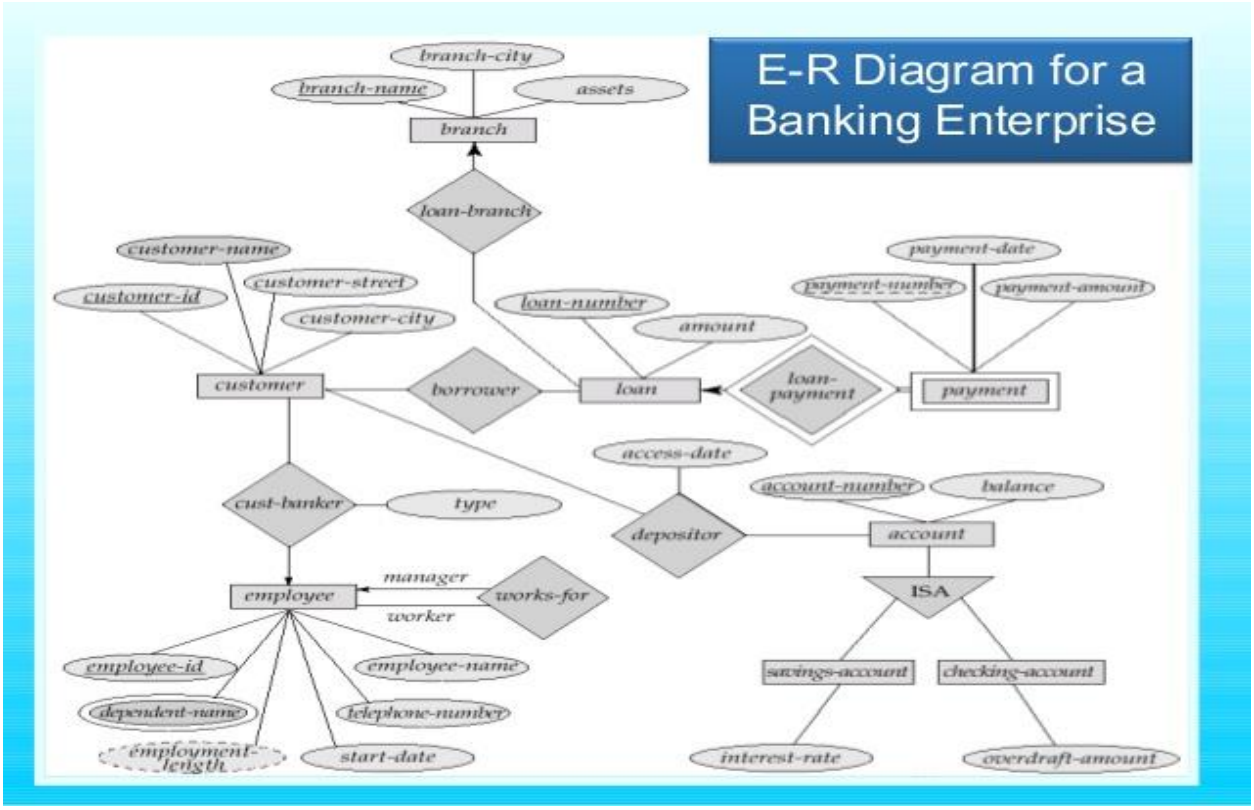


Figure 2.7 E-R diagram of Banking Enterprise System

2.10 Structure of relational Databases

The Relational Database Model

Perhaps the simplest approach to data modelling is offered by the *Relational Database Model*, proposed by Dr. Edgar F. Codd of IBM in 1970. The model was subsequently expanded and refined by its creator and very quickly became the main focus of practically all research activities in databases. The basic relational model specifies a data structure, the so-called *Relation*, and several forms of high-level languages to manipulate relations.

The term *relation* in this model refers to a two-dimensional table of data. In other words, according to the model, information is arranged in columns and rows. The term *relation*, rather than matrix, is used here because data values in the table are not necessarily homogenous (ie. Not all of the same type as, for example, in matrices of integers or real numbers). More specifically, the values in any row are not homogenous. Values in any given column, however, are all of the same type (see figure 2.8).

Customer			
	C#	Cname	Ccity	Cphone
	1	Codd	London	2263035
Row	2	Martin	Paris	5555910
	3	Deen	London	2234391

Column

Figure 2.8 Example of Relation

A relation has a unique name and represents a particular entity. Each row of a relation, referred to as a *tuple*, is a collection of facts (values) about a particular individual of that entity. In other words, a tuple represents an *instance* of the entity represented by the relation.

Features of Relational database model:

- Data is presented as a collection of relations
- Each relation is depicted as a table
- Columns are attributes
- Rows ("tuples") represent entities
- Every table has a set of attributes that taken together as a "key" (technically, a "superkey") uniquely identifies each entity

For example, a company might have an Employee table with a row for each employee. What attributes might be interesting? This, of course, depends on the application and use the data will be put to, and is determined at database design time. In our example, we might have a payroll application and need salary and mailing address information.

Table 2.1: Employee table

Employee Table

Name	SSN	StrAddr	City	State	Zip	Salary
jbs	010-00-1111	Sitterson Hall	Chapel Hill	NC	27599	120000
wms	033-53-3902	P.O.Box 3	Binghampton	NY		60000
lkb	037-84-7667	32 Juniper Rd	Bethel	CT	06801	100000
dkb	101-23-5679	1 Vegetarian Dr	Veggie	RI	21218	30000
jbs	505-47-8901	12 Onion Rd	Garlic	PU	90909	1000

Just as a side note, the notion of *view* can be useful. Imagine that a company maintains a database of its employees -- there might be a lot of attributes like age, salary, emergency contacts, appraisal, etc. There may be needs to look at the database for different applications serving different users. The company may need to make available demographic data, for example, to a governmental agency. Only some of the attributes need to be supplied - and others ought not to so as to protect privacy. Different views can be provided into the same data.

Basic structure of the Relational model:

A relation r over collection of sets (domain values) $D_1;D_2;,,,,,D_n$ is a subset of the Cartesian product $D_1X\ D_2X\ldots D_n$

A relation thus is a set of n-tuples (d1; d2; …..; dn) where
di € Di.

Given the sets

StudId = {412, 307, 540}

StudName = {Smith, Jones}

Major = {CSE, IT, BIO}

then r = {(412, Smith, CSE), (307, Jones, IT), (412, Smith, IT)} is a relation over StudId

XStudName XMajor

Relation Schema, Database Schema, and Instances

Let A_1; A_2; : : : ; A_n be attribute names with associated domains D_1; D_2; . . . , D_n, then

$$R(A_1 : D_1; A_2 : D_2; : : : ; A_n : D_n)$$

is a relation schema. For example,

Student(StudId : integer, StudName : string, Major : string)

2.11 Relational Algebra

As we have mentioned, relational algebra is a procedural language, based on algebraic concepts. It consists of a collection of operators that are defined on relations, and that produce relations as results. In this way, we can construct expressions that involve more than one operator, in order to formulate complex queries. In the following sections, we examine the various operators:

2.11.1 Fundamental Operations:

Select Operation

Notation: ϬP (r)

Defined as

ϬP (r) = { t | t € r and P(t) }

where

r is a relation (name),

P is a formula in propositional calculus, composed of conditions of the form

<attribute> = <attribute> or <constant>

Instead of "=" any other comparison predicate is allowed (≠; <;> etc).

Conditions can be composed through ∧ (and), V (or), ¬ (not)

Example: given the relation r

Example: given the relation r1

Table 2.2: r1 table

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

Ϭ $_{A=B \wedge D>5}$ (r1)

Table 2.3: modified r1 table

A	B	C	D
α	α	1	7
β	β	23	10

Project Operation

Notation: $\pi_{A1,A2,\ldots,Ak}(r)$

where A_1; : : : ; A_k are attribute names and r is a relation (name).

The result of the projection operation is defined as the relation that has k columns obtained by erasing all columns from r that are not listed.

Duplicate rows are removed from result because relations are sets.

Example: given the relations r2

Table 2.4: r2 table

A	B	C
α	10	2
α	20	2
β	30	2
β	40	4

$\Pi_{A, C} (r)$

Table 2.5: modified r2 table

A	C
α	2
α	2
β	2
β	4

Cartesian Product

Notation: r X s where both r and s are relations Defined as

$r X s := \{ t_q \mid t \in r \text{ and } q \in s \}$

Assume that attributes of r(R) and s(S) are disjoint, i.e.,

$R \cap S = \phi$

If attributes of r(R) and s(S) are not disjoint, then the rename operation must be applied first.

Example: relations r3; s1:

Table 2.6: r3 table

A	B
α	1
β	2

Table 2.7: s1 table

C	D	E
α	10	+
β	10	+
β	20	-

Table 2.8: r3 X s1 table

A	B	C	D	E
α	1	α	10	+
α	1	β	10	+
α	1	β	20	-
β	2	α	10	+

β	2	β	10	+
β	2	β	20	-

Union Operation

Notation: r U s where both r and s are relations Defined as r U s := { t | t €

r or t € s }

For r U s to be applicable,

1. r; s must have the same number of attributes
2. Attribute domains must be compatible (e.g., 3rd column of r has a data type matching the data type of the 3rd column of s)

Example: given the relations r4 and s2

Table 2.9: r4 table

A	B
α	1
α	2
β	1

Table 2.10: s2 table

A	B
α	2
β	3

Table 2.11: r4 U s2 table

A	B
α	1
α	2
β	1
β	3

Set Difference Operation

Notation: r - s where both r and s are relations

Defined as r - s := { t | t € r or t ∉ s }

For r - s to be applicable,

1. r; s must have the the same arity
2. Attribute domains must be compatible (e.g., 3rd column of r has a data type matching the data type of the 3rd column of s)

Example: given the relations r5 and s3

Table 2.12: r5 table

A	B
α	1
α	2
β	1

Table 2.13: s3 table

A	B
α	2
β	3

Table 2.14: r5 – s3 table

A	B
α	1
β	1

Rename Operation

Allows to name and therefore to refer to the result of relational algebra expression.
Allows to refer to a relation by more than one name (e.g., if the same relation is used twice in a relational algebra expression).
Example:
$\rho_x(E)$
returns the relational algebra expression E under the name x
If a relational algebra expression E (which is a relation) has
the arity k, then
$\rho_x(A_1, A_2…., A_k)(E)$
returns the expression E under the name x, and with the attribute names $A_1, A_2…., A_k$.

2.11.2 Additional Operations

These operators do not add any power (expressiveness) to the relational algebra but simplify common (often complex and lengthy) queries.
Set-Intersection ∩
Natural Join ⋈
Condition Join $\bowtie_C$ (also called Theta-Join)
Division ÷
Assignment ←

Set-Intersection

Notation: r ∩ s where both r and s are relations

Defined as r ∩ s := { t | t € r and t € s }

For r ∩ s to be applicable,

1. r; s must have the the same arity
2. Attribute domains must be compatible (e.g., 3rd column of r has a data type matching the data type of the 3rd column of s)

Derivation : r ∩ s = r – (r – s)

Example: given the relations r6 and s4

Table 2.15: r6 table

A	B
α	1
α	2
β	1

Table 2.16: s4 table

A	B
α	2
β	3

Table 2.17: r6 ∩ s4 table

A	B
α	2

Natural Join

Notation: r ⋈ s

Let r, s be relations on schemas R and S, respectively. The result is a relation on schema R U S. The result tuples are obtained by considering each pair of tuples t_r € r and t_s € s.

If t_r and t_s have the same value for each of the attributes in

R ∩ S (“same name attributes"), a tuple t is added to the result such that

t has the same value as t_r on r

t has the same value as t_s on s

Example: Given the relations R(A,B,C) and S(C,D)

Join can be applied because R ∩ S ≠ ϕ ;

the result schema is (A,B,C,D)

and the result of r ⋈ s is defined as

π r.A, r.B, r.C, r.D, s.E($\sigma_{r.B=s.B \wedge r.D=s.D}$(r X s))

Example: given the relations r7 and s5

Table 2.18: r7 table

A	B	C
p	1	a
q	2	b
p	1	c

Table 2.19: s5 table

C	D
a	11
b	21
d	12

Table 2.20: r6 ⋈ s4 table

A	B	C	D
p	1	a	11
q	2	b	21

Condition Join

Notation: $r \bowtie_C s$

C is a condition on attributes in R U S, result schema is the same as that of Cartesian Product. If $R \cap S \neq \phi$ and condition C refers to these attributes, some of these attributes must be renamed.

Sometimes also called Theta Join : ($r \bowtie_\Theta s$).

Derivation: $r \bowtie_C s = \sigma_C(r \times s)$

Note that C is a condition on attributes from both r and s

Example: given two relations r7, s5

Table 2.21: r7 table

A	B	C
1	2	3
4	5	6
7	8	9

Table 2.22: s5 table

D	E
3	1
6	2

Table 2.23: r7 $\bowtie_{B<D}$ s5

A	B	C	D	E
1	2	3	3	1
1	2	3	6	2
4	5	6	6	2

If C involves only the comparison operator "=", the condition join is also called Equi-Join.

Division

Notation: $r \div s$

Precondition: attributes in S must be a subset of attributes in R, i.e., $S \subseteq R$. Let r, s be relations on schemas R and S, respectively, where

R(A1, ……,Am, B1, ……,Bn)

S(B1,….,Bn)

The result of $r \div s$ is a relation on schema

R - S = (A1, …..,Am)

Suited for queries that include the phrase "for all".

The result of the division operator consists of the set of tuples from r defined over the attributes R - S that match the combination of every tuple in s.

$r \div s := \{t \mid t \in \pi_{R-S}(r) \wedge \forall u \in s: tu \in r\}$

Example: given the relations r8, s6:

Table 2.24: r8 table

A	B	C	D	E
1	A	1	A	1
1	A	3	A	1
1	A	3	B	1
2	A	3	A	1
2	A	3	B	3

3	A	3	A	1
3	A	3	B	1
3	A	2	B	1

Table 2.25: s6 table

D	E
A	1
B	1

Table 2.26: r8 ÷ s6 table

A	B	C
1	A	3
3	A	3

Assignment

Operation (←) that provides a convenient way to express complex queries.

Idea: write query as sequential program consisting of a series of assignments followed by an expression whose value is “displayed" as the result of the query.

Assignment must always be made to a temporary relation variable.

The result to the right of ← is assigned to the relation variable on the left of the ←. This variable may be used in subsequent expressions.

Modifications of the Database

The content of the database may be modified using the operations insert, delete or update.

Operations can be expressed using the assignment operator.

r_{new} ← operations on(r_{old})

Insert

Either specify tuple(s) to be inserted, or write a query whose result is a set of tuples to be inserted.

r ← r U E, where r is a relation and E is a relational algebra expression.

STUDENTS ← STUDENTS U {(1024, 'Clark', 'CSE', 26)}

Delete

Analogous to insert, but - operator instead of U operator.

Can only delete whole tuples, cannot delete values of particular attributes.

STUDENTS ← STUDENTS - ($\sigma_{major='CS'}$(STUDENTS))

Update

Can be expressed as sequence of delete and insert operations. Delete operation deletes tuples with their old value(s) and insert operation inserts tuples with their new value(s).

Outer Joins

Theta Join and Natural Join are called inner joins. An inner join includes only those tuples with matching attributes and the rest are discarded in the resulting relation. Therefore, we need to use outer joins to include all the tuples from the participating relations in the resulting relation. There are three kinds of outer joins − left outer join, right outer join, and full outer join.

Left Outer Join(R ⟕ S)

All the tuples from the Left relation, R, are included in the resulting relation. If there are tuples in R without any matching tuple in the Right relation S, then the S-attributes of the resulting relation are made NULL.

Table 2.27: Student table

C_id	Student_name
101	Smith
102	Alex

Table 2.28: Course table

C_id	Course_name
101	DBMS
103	OS

Table 2.29 : Student ⟕ Course

C_id	Student_name	C_id	Course_name
101	Smith	101	DBMS
102	Alex	Null	null

Right Outer Join: (R ⟖ S)

All the tuples from the Right relation, S, are included in the resulting relation. If there are tuples in S without any matching tuple in R, then the R-attributes of resulting relation are made NULL.

Table 2.30: Student table

C_id	Student_name
101	Smith
102	Alex

Table 2.31: Course table

C_id	Course_name
101	DBMS
103	OS

Table 2.32 : Student ⟖ Course

C_id	Student_name	C_id	Course_name
101	Smith	101	DBMS
Null	null	103	OS

Full Outer Join: (R ⟗ S)

All the tuples from both participating relations are included in the resulting relation. If there are no matching tuples for both relations, their respective unmatched attributes are made NULL.

Table 2.33: Student table

C_id	Student_name
101	Smith
102	Alex

Table 2.34: Course table

C_id	Course_name
101	DBMS
103	OS

Table 2.35 : Student ⋈ Course

C_id	Student_name	C_id	Course_name
101	Smith	101	DBMS
102	Alex	Null	null
Null	null	103	OS

2.12 Relational Calculus

The term *relational calculus* refers to a family of query languages, based on first order predicate calculus. These are characterized by being *declarative,* meaning that the query is specified in terms of the property of the result, rather than the procedure to be followed to obtain it. By contrast, relational algebra is known as a *procedural* language, because its expressions specify (by means of the individual applications of the operators) the construction of the result step by step.

There are many versions of relational calculus and it is not possible to present them all here. We first illustrate the version that is nearest to predicate calculus, *domain relational calculus,* which presents the basic characteristics of these languages. We then discuss the limitations and modifications that make it of practical interest. We will therefore present *tuple calculus with range declarations,* which forms the basis for many of the constructs available for queries in SQL.

In keeping with the topics already discussed concerning the relational model, we use non-positional notation for relational calculus.

This section (on calculus) and the following one (on Datalog) can be omitted without impairing the understanding of the rest of the book.

It is not necessary to be acquainted with first order predicate calculus in order to read this section. We give now some comments that enable anyone with prior knowledge to grasp the relationship with first order predicate calculus; these comments may be omitted without compromising the understanding of subsequent concepts.

There are some simplifications and modifications in relational calculus, with respect to first order predicate calculus. First, in predicate calculus, we generally have predicate symbols (interpreted in the same way as relations) and function symbols (interpreted as functions). In relational calculus, the predicate symbols correspond to relations in the database (apart from other standard predicates such as equality and inequality) and there are no function symbols. (They are not necessary given the flat structure of the relations.)

Then, in predicate calculus both open formulas (those with free variables), and closed formulas (those whose variables are all bound and none free), are of interest. The second type have a truth value that, with respect to an interpretation, is fixed, while the first have a value that depends on the values substituted for the free variables. In relational calculus, only the open formulas are of interest, A query is defined by means of an open calculus formula and the result

consists of tuples of values that satisfy the formula when substituted for free variables.

2.12.1 Domain relational calculus

Relational calculus expressions have this form:

$$\{A1:x1,...,A_k:x_k|f\}$$

where:

- $A_1,\ldots,A_k$ are distinct attributes (which do not necessarily have to appear In the schema of the database on which the query is formulated);
- $x_1,...,x_k$ are *variables* (which we will take to be distinct for the sake of convenience, even if this is not strictly necessary);
- *f* is a formula, according to the following rules:
- There are two types of *atomic* formula:
- $R(A_l:x_l. \ldots. A_PX_P)$. where $R(A_1, ..., A_p)$ is a relational schema and $x_1,..., x_p$ are variables;
- $x \upsilon y$ or $x \upsilon c$, with x and *y* variables, *c* constant and υ *comparison operator* (=, ≠. ≤, ≥, >, <).
- If f_1, and f_2 are formulas, thcn $f_1 \vee f_2$, $f_1 \wedge f_2$ and $\neg f_1$ are formulas ($\vee,\wedge,\neg$ are the *logical connectives):* where necessary, in order to ensure that the precedence is unambiguous, brackets can be used;
- If *f* is a formula and *x* a variable (which usually appears in *f*, even if not strictly necessary) then $\exists x(f)$ and $\forall x(f)$ are formulas ($\exists$ and $\forall$ are the *existential quantifier* and *universal quantifier,* respectively).

The list of pairs $A_1:x_1$,...., $A_k: x_k$ is called the *target list* because it defines the structure of the result, which is made up of the relation on $A_1,...,A_k$ that contains the tuples whose values when substituted for $x_1,...,x_k$ render the formula true. The formal definition of the truth value of a formula goes beyond the scope of this book and, at the same time, its meaning can be explained informally. Let us briefly follow the syntactic structure of formulas (the term 'value' here means 'an element of the domain', where we assume, for the sake of simplicity, that all attributes have the same domain):

- an atomic formula $R(A_1:x_1,....A_P:x_P)$ is true for values of $x_1,...,x_p$ that form a tuple of *R;*
- an atomic formula xυy is true for values of x and *y* such that the value of *x* stands in relation υ with the value of *y.* similarly for $x \upsilon c$*;*
- the meaning of connectives is the usual one;
- for the formulas built with quantifiers:
- $\exists x(f)$ is true if there exists at least one value for *x* that makes *f* true;
- $\forall x(f)$ is true if *f* is true for all possible values for *x.*

Let us now illustrate relational calculus by showing how it can be used to express the queries that we formulated in relational algebra, over the schema:

EMPLOYEES(Number, Name, Age, Salary); SUPERVISION(Head, Employee)

Let us begin with a very simple query: find the registration numbers, names, ages and salaries of the employees earning more than 40 thousand, which we can formulate in algebra with a selection:

$$\sigma_{\text{Salary} > 40}(\text{EMPLOYEES})$$

There is an equally simple formulation in relational calculus, with the expression:

$$\{\text{Number:m, Name:n, Age:a, Salary:s} \mid \text{EMPLOYEES(Number:m, Name:n, Age:a, Salary:s)} \wedge s > 40\}$$

Note the presence of two conditions in the formula (connected by the logical operator *and):*

- The first, EMPLOYEES(Number:m, Name:n, Age:a, Salary:s). requires that the values substituted respectively for the variables *m. n, a, s* constitute a tuple of the relation EMPLOYEES;
- the second requires that the value of the variable *s* is greater than 40.

The result is made up of the values of the four variables that originate from the tuples of EMPLOYEES for which the value of the salary is greater than 40 thousand.
A slightly more complex query is: *find the registration numbers, names and ages of the employees who earn more than 40 thousand.* This query requires a subset of the attributes of EMPLOYEES and thus in algebra can be formulated with a projection:

$$\pi_{Number,Name,Age}\ (\sigma_{Salary>40}(EMPLOYEES))$$

This query in calculus can be formulated in various ways. The most direct, if not the simplest, is based on the observation that what interests us are the values of Number, Name and Age, which form part of the tuples for which Salary is greater than 40. That is, for which there exists a value of Salary, greater than 40, which allows the completion of a tuple of the relation EMPLOYEES. We can thus use an existential quantifier:

$$\{Number{:}m,\ Name{:}n,\ Age{:}a \mid \exists s(EMPLOYEES(Number{:}m.\ Name{:}n.\ Age{:}a,\ Salary{:}s) \wedge s > 40)\}$$

The use of the quantifier is not actually necessary, since by simply writing

$$\{Number{:}m,\ Name{:}n,\ Age{:}a \mid EMPLOYEES(Number{:}m,\ Name{:}n,\ Age{:}a,\ Salary{:}s) \wedge s{>}40\}$$

we can obtain the same result.
The same structure can be extended to more complex queries, which in relational algebra we formulated using the join operator. We will need more atomic conditions, one for each relation involved, and we can use repeated variables to indicate the join conditions. For example, the query that requests *find the registration numbers of the supervisors of the employees who earn more than 40 thousand,* formulated in algebra:

$$\pi_{Head}(SUPRVISION \triangleright\!\triangleleft_{Employee=Number}(\sigma_{Salary>40}(EMPLOYEES)))$$

can be formulated in calculus by:

$$\{Head{:}h \mid EMPLOYEES(Number{:}m,\ Name{:}n,\ Age{:}a,\ Salary{:}s) \wedge SUPERVISION(Employee{:}\ m,\ Head{:}h) \wedge s > 40\}$$

where the variable *m.* common to both atomic conditions, builds the same correspondence between tuples specified in the join. Here, also, we can use existential quantifiers for all the variables that do not appear in the target list. However, as in the case above, this is not necessary, and would complicate the formulation.
If the involvement of different tuples of the same relation is required in an expression, then it is sufficient to include more conditions on the same predicate in the formula, with different variables. Consider the query: *find the names and salaries of the supervisors of the employees earning more than 40 thousand,* expressed in algebra, which has a join of the relation with itself:

$$\pi_{NameH,SalaryH}(\rho_{NumberH,NameH,SalaryH.AgeH \leftarrow Number,Name.Salary,Age}(EMPLOYEES) \triangleright\!\triangleleft_{NumberH=Head} (SUPERVISION \triangleright\!\triangleleft_{Employee=Number}(EMPLOYEES)))$$

This query is formulated in calculus by requiring, for each tuple of the result, the existence of three tuples: one relating to an employee earning more than 40 thousand, a second that indicates who is his supervisor, and the last (again in the EMPLOYEES relation) that gives detailed information on the supervisor:

$$\{\text{NameH:nh, SalaryH:sh} \mid \text{EMPLOYEES(Number:m, Name:n, Age:a, Salary :s)} \wedge s > 40 \wedge \text{SUPERVISION(Empioyee:m, Head:h)} \wedge \text{EMPLOYEES(Number:h, Name: nh, Age:ah, Salary :sh)}\}$$

Consider next the query: find the employees earning more than their respective supervisors, showing registration number, name and salary of the employees and supervisors (Expression in algebra). This differs from the preceding one only in the necessity of comparing values of the same attribute originating from different tuples, which causes no particular problems:

$$\{\text{Number:m. Name:n. Salary:s, NumberH:h, NameH:nh, SalaryH:sh} \mid \text{EMPLOYEES(Number:m. Name:n, Age:a, Salary:s)} \wedge \text{SUPERVISION(Employee:m, Head:h)} \wedge \text{EMPLOYEES(Number:h, Name:nh, Age:ah, Salary:sh)} \wedge s > sh\}$$

The last example requires a more complex solution. We must *find the registration numbers and names of the supervisors whose employees all earn more than 40 thousand.* In algebra we used a difference that generates the required set by taking into account all the supervisors except those who have at least one employee earning less than 40 thousand:

$$\pi_{\text{Number,Name}}(\text{EMPLOYEES} \rhd\lhd_{\text{NumberH=Head}} (\pi_{\text{Head}}(\text{SUPERVISION}) - (\pi_{\text{Head}}(\text{SUPERVISION} \rhd\lhd_{\text{Employee=Number}}(\sigma_{\text{Salary} \leq 40}(\text{EMPLOYEES})))))$$

In calculus, we must use a quantifier. By taking the same steps as for algebra, we can use a negated existential quantifier. We use many of these, one for each variable involved.

$$\{\text{Number:h, Name:n} \mid \text{EMPLOYEES(Number:h, Name:n, Age:a, Salary:s)} \wedge \text{SUPERVISION(Employee:m, Head:h)} \wedge \neg\exists m'(\exists n'(\exists a'(\exists s'(\text{EMPLOYEES(Number:m', Name:n', Age:a', Salary:s')} \wedge \text{SUPERVISION(Employee: m', Head:h)} \wedge s' <= 40)))))\}$$

As an alternative, we can use universal quantifiers:

$$(\text{Number:h, Name:n I EMPLOYEE(Number:h, Name:nh, Age:a, Salary:s)} \wedge \text{SUPERVISION(Employee:m, Head:h)} \Lambda \forall m'(\forall n'(\forall a'(\forall s'(\neg(\text{EMPLOYEES(Number:m', Name:n', Age:a', Salary:s')} \wedge \text{SUPERVISION(Employee:m', Head:h'))} \vee s' > 40)))))\}$$

This expression selects a supervisor *h* if for every quadruple of values *m', n', a',* s' relative to the employees of *h, s'* is greater than 40. The structure $\neg f \vee g$ corresponds to the condition 'If *f* then *g* (in our case, if *m'* is an employee having h as a supervisor, then the salary of *m'* is greater than 40), given that it is true in all cases apart from the one in which f is true and g is false.

It is worth noting that variations of de Morgan laws valid for Boolean algebra operators, such that:

$$\neg(f \vee g) = \neg f \wedge \neg g$$

$$\neg(f \wedge g) = \neg f \vee \neg g$$

are also valid for quantifiers:

$$\exists x(f) = \neg(\forall x(\neg (f)))$$

$$\forall x(f) = \neg(\exists x(\neg (f)))$$

The two formulations shown for the last query can be obtained one from the other by means of these equivalences. Furthermore, in general, we can use a reduced form of calculus (but without losing expressive power), in which we have the negation, a single connective (for example, the conjunction) and a single quantifier (for example the existential, which is easier to

understand).

Qualities and drawbacks of domain calculus

As we have shown in the examples, relational calculus presents some interesting aspects, particularly its declarative nature. There are, however, some defects and limitations, which are significant from the practical point of view.

First, note that calculus allows expressions that make very little sense. For example, the expression:

$$\{A_1: x_1, A_2 : x_2 \mid \mathbf{R}(\mathbf{A}_1{:}x_1) \wedge x_2 = x_2\}$$

produces as a result a relation on A_1 and A_2 made up of tuples whose values in A_1appear in the relation *R.* and the value on A_2 is any value of the domain (since the condition $x_2 = x_2$ is always true). In particular, if the domain changes, for example, from the integers between 0 and 99 to the integers between 0 and 999 the answer to the query also changes. If the domain is infinite, then the answer is also infinite, which is undesirable. A similar observation can be made for the expression

$$\{A_1: x_1 \mid\neg \mathbf{R}(A_1{:}x_1)\}$$

the result of which contains the values of the domain not appearing in *R.*

It is useful to introduce the following concept here: an expression of a query language is *domain independent* if its result, on each instance of the database, does not vary if we change the domain on the basis of which the expression is evaluated. A language is *domain independent* if all its expressions are domain independent. The requirement of domain independence is clearly fundamental for real languages, because domain dependent expressions have no practical use and can produce extensive results.

Based on the expressions seen above, we can say that relational calculus is not domain independent. At the same time, it is easy to see that relational algebra is domain independent, because it constructs the results from the relations in the database, without ever referring to the domains of the attributes. So the values of the results all come from the instance to which the expression is applied.

If we say that two query languages are *equivalent* when for each expression in one there exists an equivalent expression in the other and vice versa, we can state that algebra and calculus are not equivalent. This is because calculus, unlike algebra, allows expressions that are domain dependent. However, if we limit our attention to the subset of relational calculus made up solely of expressions that are domain independent, then we get a language that is indeed equivalent to relational algebra. In fact:

- for every expression of relational calculus that is domain independent there exists an expression of relational algebra equivalent to it;
- for every expression of relational algebra there is an expression of relational calculus equivalent to it (and thus domain independent).

The proof of equivalence goes beyond the scope of this text, but we can mention its basic principles. There is a correspondence between selections and simple conditions, between projection and existential quantification, between join and conjunction, between union and disjunction and between difference and conjunction associated with negation. The universal quantifiers can be ignored in that they can be changed to existential quantifiers using de Morgan's laws.

In addition to the problem of domain dependence, relational calculus has another disadvantage, that of requiring numerous variables, often one for each attribute of each relation involved. Then, when quantifications are necessary the quantifiers are also multiplied. The only practical languages based at least in part on domain calculus, known as *Query-by-Example (QBE),* use a graphic interface that frees the user from the need to specify tedious details.

In order to overcome the limitations of domain calculus, a variant of relational calculus has been proposed, in which the variables denote tuples instead of single values. In this way, the number of variables is often significantly reduced, in that there is only a variable for each

relation involved. This *tuple relational calculus* would however be equivalent to domain calculus, and thus also have the limitation of domain dependence. Therefore, we prefer to omit the presentation of this language. Instead we will move directly to a language that has the characteristics of tuple calculus, and at the same time overcomes the defect of domain dependence, by using the direct association of variables with relations of the database. The following section deals with this language.

2.12.2 Tuple calculus with range declarations

The expressions of tuple calculus with range declarations have the form

$$\{T|L|f\}$$

where:

- L is the *range list*, enumerating the free variables of the formula *f* with the respective ranges of variability: in fact. L is a list of elements of type *x(R)* with *x* variable and *R* relation name; if *x(R)* is in the range list, then, when the expression is evaluated, the possible values for *x* are just the tuples in the relation *R:*
- T is the *target list*, composed of elements of type *Y:x.Z* (or simply *x.Z*, abbreviation for *Z:x.Z)*, with *x* variable and *Y* and *Z* sequences of attributes (of equal length); the attributes in *Z* must appear in the schema of the relation that makes up the range of *x*. We can also write *x.**, as abbreviation for *X:x.X*, where the range of the variable χ is a relation on attributes *X;*
- *f* is a formula with
- atoms of type $x.A\upsilon c$ or $x_1.A_1\upsilon x_2.A_2$. which compare, respectively, the value of *x* on the attribute *A* with the constant *c* and the value of x_1, on A_1 with that of x_2 on A_2,
- connectives as for domain calculus;
- quantifiers, which also associate ranges to the respective variables
  $\exists x(R)(f)$
  $\forall x(R)(f)$
  where, $\exists x(R)(f)$ means 'there is a tuple *x* in the relation *R* that satisfies the formula *f*' and $\forall x(R)(f)$ means 'every tuple *x* in *R* satisfies *f*'.

Range declarations in the range list and in the quantifications have an important role: while introducing a variable *x*, a range declaration *R(x)* specifies that x can assume as values only the tuples of the relation *R* with which it is associated. Therefore this language has no need of atomic conditions such as those seen in domain calculus, which specify that a tuple belongs to a relation.

We show next how the various queries that we have already expressed in algebra and domain calculus can be formulated in this language.

The first query, which requests registration numbers, names, ages and salaries of the employees earning more than 40 thousand, becomes very concise and clear:

$$\{e.*|c(EMPLOYEES)|e.Salary>40\}$$

In order to produce only some of the attributes, *registration numbers, names and ages of the employees earning more than 40 thousand*, it is sufficient to modify the target list:

$$\{e(Number, Name, Age) \mid c(EMPLOYEES) \mid e.Salary > 40\}$$

For queries involving more than one relation, more variables are necessary, specifying the conditions of correlation on the attributes. The query that requests *find the registration numbers of the supervisors of the employees earning more than 40 thousand* can be formulated with:

$$\{s.Head \mid e(employees), s(supervision) \mid e.Number = s.Employee \wedge e.Salary > 40\}$$

Note how the formula allows for the conjunction of two atomic conditions, one that corresponds to the join condition (e.Number = s.Employee) and the other to the usual selection condition (e.Salary > 40).

In the case of expressions that correspond to the join of a relation with itself, there will be more variables with the same range. The query: *find names and salaries of supervisors of employees earning more than 40 thousand* can be formulated using the following expression:

$$\{\text{NameH, SalaryH:e'.(Name. Salary)} \mid e'(\text{EMPLOYEES}), s(\text{SUPERVISION}), e(\text{EMPLOYEES}) \mid e'.\text{Number} = s.\text{Head} \wedge s.\text{Employee} = e.\text{Number} \wedge c.\text{Salary} > 40\}$$

Similarly, we can find the employees who earn more than their respective supervisors, snowing registration number, name and salary of the employees and supervisors:

$$\{e.(\text{Name,Number,Salary}), \text{NameH, NumberH,SalaryH: e'.(Name,Number,Salary)} \mid e(\text{employees}), s(\text{supervision}), e'(\text{employees}) \mid e.\text{Number} = s.\text{Employee} \wedge s.\text{Head} = e'.\text{Number} \wedge e.\text{Salary} > e'.\text{Salary}\}$$

Queries with quantifiers are much more concise and practical here than in domain calculus. The query that requests *find the registration number and name of the supervisors whose employees all earn more that 40 thousand* can be expressed with far fewer quantifiers and variables. Again, there are various options, based on the use of the two quantifiers and of negation. With universal quantifiers:

$$\{e.(\text{Number, Name}) \mid e(\text{EMPLOYEES}), s(\text{SUPERVISION}) \mid e.\text{Number} = s.\text{Head} \wedge \forall e''(\text{EMPLOYEES})(\forall s'(\text{SUPERVSION}) (\neg(s.\text{Head} = s'.\text{Head} \wedge s'.\text{Employee} = e'.\text{Number}) \vee e'.\text{Salary} > 40))\}$$

With negated existential quantifiers:

$$\{e.(\text{Numbcr, Name}) \mid e(\text{EMPLOYEES}), s(\text{SUPERVISION}) \mid e.\text{Number} = s.\text{Head} \wedge \neg(\exists e'(\text{EMPLOYEES}) (\exists s'(\text{SUPERVISION}) (s.\text{Head} = s'.\text{Head} \wedge s'.\text{Employee} = e'.\text{Number} \wedge e'.\text{Salary} \leq 0)))\}$$

Unfortunately, it turns out that it is not possible in tuple calculus with range declarations to express all the queries that could be formulated in relational algebra (or in domain calculus). In particular, the queries that in algebra require the union operator cannot be expressed in this version of calculus. Take, for example, the simple union of two relations on the same attributes: given $R_1(AB)$ and $R_2(AB)$, we wish to formulate the query that we would express in algebra by the union of R_1 and R_2. If the expression had two free variables, then every tuple of the result would have to correspond to a tuple of each of the relations. This is not necessary, because the union requires the tuples of the result to appear in at least one of the operands, not necessarily in both. If, on the other hand, the expression had a single free variable, this would have to refer to a single relation, without acquiring tuples from the other for the result. Therefore, the union cannot be expressed.

For this reason, SQL, allows for an explicit union construct, to express queries that would otherwise prove impossible. This is because the declarative aspects of SQL are based on tuple calculus with range declarations.

2.13 Views

In Chapter 1, we saw how it can be useful to make different representations of the same data available to users. In the relational model, this is achieved by means of *derived relations*, that is, relations whose content is defined in terms of the contents of other relations. Thus, in a relational database there can exist *base* relations, whose content is autonomous and actually stored in the database, and derived relations, whose content is derived from the content of other relations. It is possible that a derived relation is defined in terms of other derived relations, on condition that an ordering exists among the derived relations, so that all derived relations can be expressed in terms of base relations. There are basically two types of derived relations:

• *materialized views:* derived relations that are actually stored in the database;
• *virtual relations* (also called *views,* without further qualification): relations defined by means of functions (expressions in the query language), not stored in the database, but usable in the queries as if they were.

Materialized views have the advantage of being immediately available for queries. Frequently, however, it is a heavy task to maintain their contents consistent with those of the relations from which they are derived, as any change to the base relations from which they depend has to be propagated to them. On the other hand, virtual relations must be recalculated for each query but produce no consistency problems. Roughly, we can say that materialized views are convenient when there are fewer updates than queries and the calculation of the view is complex. It is difficult, however, to give general techniques for maintaining consistency between base relations and materialized views. For this reason, most commercial systems provide mechanisms for organizing only virtual relations, which from here on, with no risk of ambiguity, we call simply *views.*

Views are defined in relational systems by means of query language expressions. Then queries on views are resolved by substituting the definition of the view for the view itself, that is, by composing the original query with the view query. For example, consider a database on the relations:

$$R_1(ABC),\ R2_2(DEF),\ R_3(GH)$$

with a view defined using a cartesian product followed by a selection

$$R=\sigma_{A>D}(R_1 \triangleright\triangleleft R_2)$$

On this schema, the query

$$\sigma_{B=G}(R \triangleright\triangleleft R_3)$$

is executed by replacing *R* with its definition

$$\sigma_{B=G}(\sigma_{A>D}(R_1 \triangleright\triangleleft R_2) \triangleright\triangleleft R_3)$$

The use of views can be convenient for a variety of reasons,

- A user interested in only a portion of the database can avoid dealing with the irrelevant components. For example, in a database with two relations on the schemas

  EMPLOYEES(Employee, Department; MANAGERS(Department, Supervisor)

- A user interested only in the employees and their respective supervisors could find his task facilitated by a view defined as follows:

$$\pi_{\text{Employee, Supervisor}}(\text{EMPLOYEES} \triangleright\triangleleft \text{MANAGERS})$$

- Very complex expressions can be defined using views, with particular advantages in the case of repeated sub-expressions.
- By means of access authorizations associated with views, we can introduce mechanisms for the protection of privacy; for instance, a user could be granted restricted access to the database through a specifically designed view.
- In the event of restructuring of a database, it can be convenient to define views corresponding to relations that are no longer present after the restructuring. In this way, applications written with reference to the earlier version of the schema can be used on the new one without the need for modifications. For example, if a schema *R(ABC) i*s replaced by two schemas $R_1(AB)$; $R_2(BC)$, *w*e can define a view, $R = R_1 \triangleright\triangleleft R_2$ *a*nd leave intact the applications that refer to *R. T*he results as we show in Chapter 8 confirm that, if *B i*s a key for R_2. *t*hen the presence of the view is completely transparent.

As far as queries are concerned, views can be treated as if they were base relations. However, the same cannot be said for update operations. In fact, it is often not even possible to define semantics for updating views. Given an update on a view, we would like to have exactly one set of updates to the base relations, such that the view, if computed after these changes to the

base relations, appears as if the given update had been performed on it. Unfortunately, this is not generally possible. For example, let us look again at the view

$$\pi_{\text{Employee, Supervisor}}(\text{EMPLOYEES} \triangleright\!\triangleleft \text{MANAGERS})$$

Assume we want to insert a tuple into the view: we would like to have tuples to insert into the base relations that allow the generation of the new tuple in the view. But this is not possible, because the tuple in the view does not involve the Department attribute, and so we do not have a value for it, as needed in order to establish the correspondence between the two relations. In general, the problem of updating views is complex, and all systems have strong limitations regarding the updating of views.

MODULE 2 MCQ and Short type Problem

Multiple choice questions (MCQ)

1. In database records are called:
a. Attributes
b. Entity
c. Tuples
d. Relations
Ans. (c)

2. An entity has a set of _____ that describe it.
a. Attributes
b. Entity
c. Tuples
d. Relations
Ans. (a)

3. In ER model rectangle represents:
a. Attributes
b. Entity set
c. Relationships
d. None of these
Ans. (b)

4. Date is the type of attribute:
a. Simple
b. Composite
c. Single values
d. Multi valued
Ans. (b)

5. _____is the attribute or group of attributes that uniquely identify occurrence of each entity.
a. Foreign key
b. Super Key
c. Primary Key
d. All of these
Ans. (b)

6. _____is the real world object, such as a person, place etc.
a. Attribute
b. Entity
c. Records

d. All of these
Ans. (b)

7. Grant and revoke is the type of command:
a. DDL
b. DML
c. DCL
d. DQL
Ans. (c)

8. A user that manages the files of application in DBMS is called:
a. Administrator
b. Database analyst
c. File Manager
d. None of these
Ans. (a)

9. _____is the information about data.
a. Data
b. Meta-Data
c. Entity
d. Relations
Ans. (b)
10. Which one is the fundamental operation?
a. Insertion b. Cartesian product c. Join d. intersection
Ans. (b)

Short question

1. What is weak entity set? Explain with example.
2. What is primary key, super key, candidate key and foreign key?
3. How additional operation can be expressed using fundamental operation? Explain.

Long type question:

1. A university registrar's office maintains data about the following entities:
    - Courses, including number, title, credits, syllabus, and prerequisites;
    - Course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom;
    - Students, including student-id, name, and program;
    - Instructors, including identification number, name, department, and title.

   Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled. Construct a E-R diagram for registrar's office. Document all assumptions that you make about the mapping constraints.
2. Design an E-R diagram for keeping track of the exploits of your favorite sports team. You should store the matches played, the scores in each match, the players in

each match, and individual player statistics for each match. Summary statistics should be modeled as derived attributes.

3 Define single valued and multi valued attribute? What does cardinality ratio specify? What are disadvantages of DBMS?

4 Explain generalization, specialization and aggregation.

5 Assume the following relations:

BOOKS(DocId, Title, Publisher, Year)
STUDENTS(StId, StName, Major, Age)
AUTHORS(AName, Address)
borrows(DocId, StId, Date)
has-written(DocId, AName)
describes(DocId, Keyword)
a) List the year and title of each book.
b) List all information about students whose major is CS.
c) List all students with the books they can borrow.
d) List all books published by McGraw-Hill before 1990.
e) List the name of those authors who are living in Davis.
f) List the name of students who are older than 30 and who are not studying CS.
g) Rename AName in the relation AUTHORS to Name.

Text Books:

1. Henry F. Korth and Silberschatz Abraham, "Database System Concepts", Mc.Graw Hill.
2. Elmasri Ramez and Novathe Shamkant, "Fundamentals of Database Systems", Benjamin Cummings Publishing. Company.
3. Date C. J., "Introduction to Database Management", Vol. I, II, III, Addison Wesley.
4. Ramakrishnan: Database Management System , McGraw-Hill
5. Ullman JD., "Principles of Database Systems", Galgottia Publication.

Module 3:

SQL and Integrity Constraints [6L]

Concept of DDL, DML, DCL. Basic Structure, Set operations, Aggregate Functions, Null Values, Domain Constraints, Referential Integrity Constraints, assertions, views, Nested Subqueries, Database security application development using SQL, Stored procedures and triggers.

3.1 Introduction about SQL-

SQL (Structured Query Language) is a nonprocedural language, you specify what you want, not how to get it. A block structured format of English key words is used in this Query language. It has the following components.

3.1.1 DDL (Data Definition Language)-

The SQL DDL provides command for defining relation schemas, deleting relations and modifying relation schema. It includes create table, alter table, drop table in the database.

3.1.2 DML (DATA Manipulation Language)-

It includes commands to insert tuples into, delete tuples from and modify tuples in the database.

3.1.3 DCL (DATA Control Language)-

A **data control language** (**DCL**) is a syntax similar to a computer programming language used to control access to data stored in a database (Authorization). In particular, it is a component of Structured Query Language (SQL).

Examples of DCL commands include:

- GRANT to allow specified users to perform specified tasks.
- REVOKE to cancel previously granted or denied permissions.

3.2 View definition-

The SQL DDL includes commands for defining views.
Transaction Control- SQL includes for specifying the beginning and ending of transactions.

Embedded SQL and Dynamic SQL-

Embedded and Dynamic SQL define how SQL statements can be embedded with in general purpose programming languages, such as C, C++, JAVA, COBOL, Pascal and Fortran.

Integrity-

The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must specify. Updates that violate integrity constraints are allowed.
Authorization-
The SQL DDL includes commands for specifying access rights to relations and views.

Data Definition Language-

The SQL DDL allows specification of not only a set of relations but also information about each relation, including-
Schema for each relation
The domain of values associated with each attribute.
The integrity constraints.
The set of indices to be maintained for each relation.
The security and authorization information for each relation.
The physical storage structure of each relation on disk.

3.3 Domain types in SQL-

The SQL standard supports a variety of built in domain types, including-
Char (n)- A fixed length character length string with user specified length .
Varchar (n)- A variable character length string with user specified maximum length n.
Int- An integer.
Small integer- A small integer.
Numeric (p, d)-A Fixed point number with user defined precision.
Real, double precision- Floating point and double precision floating point numbers with machine dependent precision.
Float (n)- A floating point number, with precision of at least n digits.
Date- A calendar date containing a (four digit) year, month and day of the month.
Time- The time of day, in hours, minutes and seconds Eg. Time '09:30:00'.
Number- Number is used to store numbers (fixed or floating point).

3.4 DDL statement for creating a table-

Syntax-
Create table tablename
(columnname datatype(size), columnname datatype(size)); Creating a table from a table-
Syntax-
CREATE TABLE TABLENAME

[(columnname, columnname, ……)]
AS SELECT columnname, columnname FROM tablename;

3.5 Insertion of data into tables-

Syntax-
INSERT INTO tablename [(columnname, columnname, ……)] Values(expression, expression);

Inserting data into a table from another table:

Syntax-
INSERT INTO tablename

SELECT columnname,
columnname, ……. FROM
tablename;

Insertion of selected data into a table from another table:

Syntax-
INSERT INTO tablename
SELECT columnname, columnname……..
FROM tablename
WHERE columnname= expression;

3.6 Retrieving of data from the tables-

Syntax-
SELECT * FROM tablename;

The retrieving of specific columns from a table-

Syntax-
SELECT columnname, columnname, …. FROM tablename;

Elimination of duplicates from the select statement-

Syntax-
SELECT DISTINCT columnname, columnname FROM tablename;

Selecting a data set from table data-

Syntax-
SELECT columnname, columnname FROM tablename
WHERE searchcondition;

3.7 Updating the content of a table:

In creation situation we may wish to change a value in table without changing all values in the tuple . For this purpose the update statement can be used.

Update table name
Set columnname = expression, columnname =expression…… Where columnname = expression;

3.8 Deletion Operation:-

A delete query is expressed in much the same way as Query. We can delete whole tuple (rows) we can delete values on only particulars attributes.

Deletion of all rows

Syntax:
Delete from tablename :

Deletion of specified number of rows Syntax:

Delete from table name Where search condition ;

Computation in expression lists used to select data

+ Addition - Subtraction
multiplication *
/ Division () Enclosed operation

Renaming columns used with Expression Lists: - The default output column names can be renamed by the user if required
Syntax:

Select column name result_columnname, Columnname result_columnname,
From table name;
Logical Operators:
The logical operators that can be used in SQL sentenced are

AND all of must be included
OR any of may be included
NOT none of could be included
Range Searching: Between operation is used for range searching.
Pattern Searching:
The most commonly used operation on string is pattern matching using the operation 'like' we describe patterns by using two special characters.

Percent (%) ; the % character matches any substring we consider the following examples.
'Perry %' matches any string beginning with perry
'% idge % matches any string containing' idge as substring.
' - - - ' matches any string exactly three characters.
' - - - % matches any string of at least of three characters.
3.9 Oracle functions:
Functions are used to manipulate data items and return result. function follow the format of function _name (argument1, argument2 ..) .An arrangement is user defined variable or constant. The structure of function is such that it accepts zero or more arguments.
Examples:
Avg return average value of n

Syntax:
Avg ([distinct/all]n)
Min return minimum value of expr.

Syntax:
MIN((distinct/all)expr)
Count Returns the no of rows where expr is not null
Syntax:
Count ([distinct/all)expr]
Count (*) Returns the no rows in the table, including duplicates and those with nulls. Max
Return max value of expr

Syntax:

Max ([distinct/all]expr)
Sum Returns sum of values of n

Syntax:
Sum ([distinct/all]n)

Sorting of data in table

Syntax:
Select columnname, columnname From table
Order by columnname;
3.10 Primary Key: primary key is one or more columns is a table used to uniquely identity each row in the table. Primary key values must not be null and must be unique across the column. A multicolumn primary key is called composite primary key.

Syntax: primary key as a column constraint
Create table tablename
(columnname datatype (size) primary key,….)
Primary key as a table constraint
Create table tablename
(columnname datatype (size), columnname datatype(size)… Primary key
(columnname,columnname));
Default value concept: At the line of cell creation a default value can be assigned to it. When the user is loading a record with values and leaves this cell empty, the DBA will automatically load this cell with the default value specified. The data type of the default value should match the data type of the column
Syntax:
Create table tablename
(columnname datatype (size) default value,….);

3.11 Foreign Key Concept : Foreign key represents relationship between tables. A foreign key is column whose values are derived from the primary key of the same of some other table. the existence of foreign key implies that the table with foreign key is related to the primary key table from which the foreign key is derived .A foreign key must have corresponding primary key value in the primary key table to have meaning.
Foreign key as a column constraint
Syntax :
Create table table name
(columnname datatype (size) references another table name);
Foreign key as a table constraint:

Syntax :
Create table name (columnname datatype (size)…. primary key (columnname);
foreign key (columnname)references table name);

3.12 Check Integrity Constraints: Use the check constraints when you need to enforce integrity rules that can be evaluated based on a logical expression following are a few examples of appropriate check constraints.
A check constraints name column of the client_master so that the name is entered in upper case.
A check constraint on the client_no column of the client _master so that no client_no value starts with ‘c’.
Syntax:
Create table tablename
(columnname datatype (size) CONSTRAINT constraintname) Check (expression));
Adding new columns: Syntax
ALTER TABLE tablename
ADD (newcolumnname newdatatype (size));
Modifying existing table Syntax:
ALTER TABLE tablename
MODIFY (newcolumnname newdatatype (size));

NOTE: Oracle not allow constraints defined using the alter table, if the data in the table, violates such constraints.

Removing/Deleting Tables-

Following command is used for removing or deleting a table.

Syntax:

DROP TABLE tablename:

Defining Integrity constraints in the ALTER TABLE command-

You can also define integrity constraints using the constraint clause in the ALTER TABLE command. The following examples show the definitions of several integrity constraints.

Add PRIMARY KEY- Syntax:

ALTER TABLE tablename
ADD PRIMARY KEY (columnname);

Add FOREIGN KEY- Syntax:

ALTER TABLE tablename
ADD CONSTRAINT constraintname
FOREIGN KEY(columnname) REFERENCES tablename; Dropping integrity constraints in the ALTER TABLE command:

You can drop an integrity constraint if the rule that if enforces is no longer true or if the constraint is no longer needed. Drop the constraint using the ALTER TABLE command with the DROP clause. The following examples illustrate the dropping of integrity constraints.

DROP the PRIMARY KEY- Syntax:

ALTER TABLE tablename DROP PRIMARY KEY

DROP FOREIGN KEY- Syntax:

ALTER TABLE tablename
DROP CONSTRAINT constraintname;

3.13 Algorithm for JOIN in SQL:

a. Cartesian product of tables (specified in the FROM clause)
b. Selection of rows that match (predicate in the WHERE clause)
c. Project column specified in the SELECT clause.

Cartesian product:-

Consider two table student and course Select B.*,P.*
FROM student B, course P;

INNER JOIN:

Cartesian product followed by selection Select B.*,P.*
FROM student B, Course P WHERE B.course # P.course # ;

LEFT OUTER JOIN:

LEFT OUTER JOIN = Cartesian product + selection but include rows from the left table which are unmatched pat nulls in the values of attributes belonging to th e second table

Exam:

Select B.*,P*
FROM student B left join course p ON B.course # P.course #;

RIGHT OUTER JOIN:

RIGHT OUTER JOIN = Cartesian product + selection but include rows from right table which are unmatched

Example:

Select B.*,P.*
From student B RIGHT JOIN course P B.course# = P course # ;

FULL OUTER JOIN
Exam
Select B.*,P.*
From student B FULL JOIN course P On B.course # = P course # ;

3.14 Grouping Data From Tables:
There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples, we specify this wish in SQL using the group by clause. The attribute or attributes given in the group by clause are used to form group. Tuples with the same value on all attributes in the group by clause are placed in one group.
Syntax:
SELECT columnname, columnname FROM tablename
GROUP BY columnname;

Using the Union, Intersect and Minus Clause:
Union Clause:
The user can put together multiple Queries and combine their output using the union clause . The union clause merges the output of two or more Queries into a single set of rows and column. The final output of union clause will be

Output: = Records only in Query one + records only in Query two + A single set of records with is common in the both Queries.
Syntax:
SELECT columnname, columname FROM tablename 1
UNION
SELECT columnname, columnname From tablename2;
Intersect Clause: The use can put together multiple Queries and their output using the interest clause. The final output of the interest clause will be :

Output =A single set of records which are common in both Queries Syntax:

SELECT columnname, columnname FROM tablename 1
INTERSECT
SELECT columnname, columnname FROM tablename 2;

MINUS CLAUSE:- The user can put together multiple Queries and combine their output
= records only in Query one Syntax:
SELECT columnname, columnname FROM tablename ;
MINUS
SELECT columnname, columnname FROM tablename ;

3.15 Indexes- An index is an ordered list of content of a column or group of columns in a table. An index created on the single column of the table is called simple index. When multiple table columns are included in the index it is called composite index.
Creating an Index for a table:-
Syntax (Simple)
CREATE INDEX index_name ON tablename(column name);
Composite Index:-
CREATE INDEX index_name
ON tablename(columnname,columnname);

Creating an UniQuestion Index:-
CREATE UNIQUESTION INDEX indexfilename ON tablename(columnname);
Dropping Indexes:-
An index can be dropped by using DROP INDEX

SYNTAX:-
DROP INDEX indexfilename;

3.16 Views:-

Logical data is how we want to see the current data in our database. Physical data is how this data is actually placed in our database.
Views are masks placed upon tables. This allows the programmer to develop a method via which we can display predetermined data to users according to our desire.
Views may be created fore the following reasons:

The DBA stores the views as a definition only. Hence there is no duplication of data.
Simplifies Queries.
Can be Queried as a base table itself.
Provides data security.
Avoids data redundancy.

Creation of Views:-

Syntax:-
CREATE VIEW viewname AS SELECT columnname,columnname FROM tablename
WHERE columnname=expression_list;

Renaming the columns of a view:-

Syntax:-
CREATE VIEW viewname AS SELECT newcolumnname…. FROM tablename
WHERE columnname=expression_list;
Selecting a data set from a view-
Syntax:-
SELECT columnname, columnname FROM viewname
WHERE search condition;

Destroying a view-

Syntax:-
DROP VIEW viewname;

3.17 GRANT Statement

The GRANT Statement grants access privileges for database objects to other users. It has the following general format:

GRANT privilege-list ON [TABLE] object-list TO user-list

privilege-list is either ALL PRIVILEGES or a comma-separated list of properties: SELECT, INSERT, UPDATE, DELETE. *object-list* is a comma-separated list of table and view names. *user-list* is either PUBLIC or a comma-separated list of user names.
The GRANT statement grants each privilege in *privilege-list* for each object (table) in *object-list* to each user in *user-list*. In general, the access privileges apply to all columns in the table or view, but it is possible to specify a column list with the UPDATE privilege specifier:

UPDATE [(column-1 [, column-2] ...)]

If the optional column list is specified, UPDATE privileges are granted for those columns only.
The *user-list* may specify PUBLIC. This is a general grant, applying to all users (and future users) in the catalog.
Privileges granted are revoked with the REVOKE Statement.

The optional specificier WITH GRANT OPTION may follow *user-list* in the GRANT statement. WITH GRANT OPTION specifies that, in addition to access privileges, the privilege to grant those privileges to other users is granted.

GRANT Statement Examples

GRANT SELECT ON s,sp TO PUBLIC

GRANT SELECT,INSERT,UPDATE(color) ON p TO art,nan

GRANT SELECT ON supplied_parts TO sam WITH GRANT OPTION

REVOKE Statement

The REVOKE Statement revokes access privileges for database objects previously granted to other users. It has the following general format:

REVOKE privilege-list ON [TABLE] object-list FROM user-list

The REVOKE Statement revokes each privilege in *privilege-list* for each object (table) in *object-list* from each user in *user-list*. All privileges must have been previously granted.
The user-list may specify PUBLIC. This must apply to a previous GRANT TO PUBLIC.

REVOKE Statement Examples

REVOKE SELECT ON s,sp FROM PUBLIC

REVOKE SELECT,INSERT,UPDATE(color) ON p FROM art,nan

REVOKE SELECT ON supplied_parts FROM sam

3.18 PL/SQL:

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database.

Following are certain notable facts about PL/SQL:

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.

Features of PL/SQL

PL/SQL has the following features:

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages:

- SQL is the standard database language and PL/SQL is strongly integrated with SQL.

- PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding
- DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time.
- This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages

Now, we will discuss the Basic Syntax of PL/SQL which is a **block-structured** language; this means that the PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts:

Sl. No.	Sections and descriptions
1	**Declarations** This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
2	**Executable Commands** This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.
3	**Exception Handling** This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

Every PL/SQL statement ends with a semicolon **(;)**. PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block:

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>
EXCEPTION
<exception handling>
END;
```

The 'Hello World' Example

```
DECLARE
message varchar2(20):= 'Hello, World!';
BEGIN
dbms_output.put_line(message);
END;
/
```

The **end;** line signals the end of the PL/SQL block. To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code. When the above code is executed at the SQL prompt, it produces the following result:

Hello World
PL/SQL procedure successfully completed.

3.19 Database Triggers

A database trigger is a stored PL/SQL program unit associated with a specific database table. ORACLE executes (fires) a database trigger automatically when a given SQL operation (like INSERT, UPDATE or DELETE) affects the table. Unlike a procedure, or a function, which must be invoked explicitly, database triggers are invoked implicitly.

Database triggers can be used to perform any of the following:

- Audit data modification
- Log events transparently
- Enforce complex business rules
- Derive column values automatically
- Implement complex security authorizations
- Maintain replicate tables

You can associate up to 12 database triggers with a given table. A database trigger has three parts: a **triggering event**, an **optional trigger constraint**, and a **trigger action**. When an event occurs, a database trigger is fired, and an predefined PL/SQL block will perform the necessary action. The owner, not the current user, must have appropriate access to all objects referenced by the trigger action.

You cannot use COMMIT, ROLLBACK and SAVEPOINT statements within trigger blocks. You have to be careful with using triggers as it may be executed thousands of times for a large update, and therefore can seriously affect SQL execution performance.

The syntax for creating a trigger (the reserved words and phrases surrounded by brackets are optional):

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE|AFTER} triggering_event ON table_name
[FOR EACH ROW]
[WHEN condition]
DECLARE
        Declaration statements
BEGIN
        Executable statements
EXCEPTION
        Exception-handling statements
END;
```

The reserved word CREATE specifies that you are creating a new trigger. The reserved word REPLACE specifies that you are modifying an existing trigger. OR REPLACE is optional.

The trigger_name references the name of the trigger.

BEFORE or AFTER specify when the trigger is fired (before or after the triggering event).
The triggering_event references a DML statement issued against the table.

The table_name is the name of the table associated with the trigger.
The clause, FOR EACH ROW, specifies a trigger is a row trigger and fires once for each modified row.
A WHEN clause specifies the condition for a trigger to be fired.
Bear in mind that if you drop a table, all the associated triggers for the table are dropped as well.

3.20 Database Cursor

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it.

This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the *active* set.

Implicit Cursor:

These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.

Explicit Cursor:

They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.

Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

Implicit Cursors: Application

When you execute DML statements like DELETE, INSERT, UPDATE and SELECT statements, implicit statements are created to process these statements.

Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

For example, When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected. When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

For Example: Consider the PL/SQL Block that uses implicit cursor attributes as shown below:

```
DECLARE  var_rows number(5);
BEGIN
  UPDATE employee
  SET salary = salary + 1000;
  IF SQL%NOTFOUND THEN
    dbms_output.put_line('None of the salaries where updated');
```

```
  ELSIF SQL%FOUND THEN
    var_rows := SQL%ROWCOUNT;
    dbms_output.put_line('Salaries for ' || var_rows || 'employees are
updated');
  END IF;
END;
```

In the above PL/SQL Block, the salaries of all the employees in the ‘employee’ table are updated. If none of the employee’s salary are updated we get a message 'None of the salaries where updated'. Else we get a message like for example, 'Salaries for 1000 employees are updated' if there are 1000 rows in ‘employee’ table.

MODULE 3 MCQ and Short type Problem

Multiple choice questions (MCQ)

1. Which one of the following is a set of one or more attributes taken collectively to uniquely identify a record?
a) Candidate key
b) Sub key
c) Super key
d) Foreign key
Answer: c
2. Consider attributes ID , CITY and NAME . Which one of this can be considered as a super key ?
a) NAME
b) ID
c) CITY
d) CITY , ID
Answer: b
3. The subset of super key is a candidate key under what condition ?
a) No proper subset is a super key
b) All subsets are super keys
c) Subset is a super key
d) Each subset is a super key
Answer: a
4. Which one of the following is used to define the structure of the relation ,deleting relations and relating schemas ?
a) DML(Data Manipulation Langauge)
b) DDL(Data Definition Langauge)
c) Query
d) Relational Schema
Answer: b
5. Which one of the following provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database ?
a) DML(Data Manipulation Langauge)
b) DDL(Data Definition Langauge)
c) Query
d) Relational Schema
Answer: a
6. Create table employee (name varchar ,id integer)
What type of statement is this ?
a) DML
b) DDL
c) View

d) Integrity constraint

Answer: b
7. Select * from employee
What type of statement is this?
a) DML
b) DDL
c) View
d) Integrity constraint
Answer: a
8. The basic data type char(n) is a _____ length character string and varchar(n) is _____ length character.
a) Fixed, equal
b) Equal, variable
c) Fixed, variable
d) Variable, equal
Answer: c
9. An attribute A of datatype varchar(20) has the value "Avi" . The attribute B of datatype char(20) has value "Reed" .Here attribute A has ____ spaces and attribute B has ____ spaces.
a) 3, 20
b) 20, 4
c) 20 , 20
d) 3, 4
Answer: a
10. To remove a relation from an SQL database, we use the _____ command.
a) Delete
b) Purge
c) Remove
d) Drop table
Answer: d

Short question

1) Explain outer join operations.
2) What is the role of cursor?
3) What is the role of trigger?

Long type question:

1) i) Consider the following relational database, where the primary keys are underlined. Give an expression in the relational algebra to express each of the following queries:

   employee(person_name,street,city)
   works(person_name,company_name,salary)
   company(company_name,city)
   manages(person_name,manager_name)

a) Find the names and cities of residence of all employees who work for First Bank Corporation.
b) Find the names, street address and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum.
c) Find the names of all employees in the database who live in the same city as the company for which they work.
d) Give all managers in this database a 10 percent salary increase.
e) Modify the database so that Jones now lives in New Town. 5X2=10

ii) Define: Super Key, Candidate Key, Primary Key, Foreign Key, Relation. 5

2. i) Consider the following relational database and answer the queries in SQL.
EMP (Empno, Ename, Job, Mgr, Hiredate, Sal, Comm, Deptno)
DEPT (Deptno, Dname, Loc)
SALGRADE (Grade, Losal, Hisal)

a) Show all employee whose salary is greater than any one's salary of department 10.

b) Show all employees who are earning the highest salary in each department.

c) Show all employees along with their manager.

d) Show the grade of the employee whose name is of exactly 4 characters. 2X4=8

ii) What are the different database users? 4
iii) Explain briefly different kinds of Outer Join. 3

Text Books:

1. Henry F. Korth and Silberschatz Abraham, "Database System Concepts", Mc.Graw Hill.
2. Elmasri Ramez and Novathe Shamkant, "Fundamentals of Database Systems", Benjamin Cummings Publishing. Company.
3. Date C. J., "Introduction to Database Management", Vol. I, II, III, Addison Wesley.

Module 4:
Relational Database Design [8L]
Functional Dependency, Different anomalies in designing a Database., Normalization using functional dependencies, Decomposition, Boyce-Codd Normal Form, 3NF, Normalization using multi-valued dependencies, 4NF, 5NF , Case Study

4.1 Introduction:

Relational database design ultimately produces a set of relations. The implicit goals of the design activity are: *information preservation and minimum redundancy*.

Informal Design Guidelines for Relation Schemas

Four *informal guidelines* that may be used as *measures to determine the quality* of relation schema design:

- Making sure that the semantics of the attributes is clear in the schema
- Reducing the redundant information in tuples
- Reducing the NULL values in tuples
- Disallowing the possibility of generating spurious tuples

4.2 Imparting Clear Semantics to Attributes in Relations

The **semantics** of a relation refers to its meaning resulting from the interpretation of attribute values in a tuple. The relational schema design should have a clear meaning.

Guideline 1

1. Design a relation schema so that it is easy to explain.
2. Do not combine attributes from multiple entity types and relationship types into a single relation.

Redundant Information in Tuples and Update Anomalies

One goal of schema design is to minimize the storage space used by the base relations (and hence the corresponding files). Grouping attributes into relation schemas has a significant effect on storage space storing natural joins of base relations leads to an additional problem referred to as **update anomalies**. These are: insertion anomalies, deletion anomalies, and modification anomalies.

Insertion Anomalies happen:

When insertion of a new tuple is not done properly and will therefore can make the database become inconsistent.

When the insertion of a new tuple introduces a NULL value (for example a department in which no employee works as of yet). This will violate the integrity constraint of the table since ESsn is a primary key for the table.

Deletion Anomalies:

The problem of deletion anomalies is related to the second insertion anomaly situation just discussed.

Example: If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database.

Modification Anomalies

Happen if we fail to update all tuples as a result in the change in a single one.

Example: if the manager changes for a department, all employees who work for that department must be updated in all the tables. It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data as well as require unnecessary updates that can be avoided

Guideline 2

Design the base relation schemas so that no insertion, deletion, or modification anomalies are present in the relations.

If any anomalies are present, note them clearly and make sure that the programs that update the database will operate correctly. The second guideline is consistent with and, in a way, a restatement of the first guideline.

4.3 NULL Values in Tuples

Fat Relations: A relation in which too many attributes are grouped. If many of the attributes do not apply to all tuples in the relation, we end up with many NULLs in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes and with specifying JOIN operations at the logical level.

Another problem with NULLs is how to account for them when aggregate operations such as COUNT or SUM are applied. SELECT and JOIN operations involve comparisons; if NULL values are present, the results may become unpredictable. Moreover, NULLs can have multiple interpretations, such as the following:

- The attribute *does not apply* to this tuple. For example, Visa_status may not apply to U.S. students.
- The attribute value for this tuple is *unknown*. For example, the Date_of_birth may be unknown for an employee.
- The value is *known but absent*; that is, it has not been recorded yet. For example, the Home_Phone_Number for an employee may exist, but may not be available and recorded yet.

Having the same representation for all NULLs compromises the different meanings they may have. Therefore, we may state another guideline.

Guideline 3

As much as possible, avoid placing attributes in a base relation whose values may frequently be NULL.

If NULLs are unavoidable, make sure that they apply in exceptional cases only.
For example, if only 15 percent of employees have individual offices, there is little justification for including an attribute
Office_number in the EMPLOYEE relation; rather, a relation
EMP_OFFICES(Essn, Office_number) can be created

Generation of Spurious Tuples

Often, we may elect to split a "fat" relation into two relations, with the intention of joining them together if needed. However, applying a NATURAL JOIN may not yield the desired effect. On the contrary, it will generate many more tuples and we cannot recover the original table.

Guideline 4

Design relation schemas so that they can be joined with equality conditions on attributes that are appropriately related (primary key, foreign key) pairs in a way that guarantees that no spurious tuples are generated.
Avoid relations that contain matching attributes that are not (foreign key, primary key) combinations because joining on such attributes may produce spurious tuples.

Summary and Discussion of Design Guidelines

We proposed informal guidelines for a good relational design. The problems we pointed out, which can be detected without additional tools of analysis, are as follows:

- Anomalies that cause redundant work to be done during insertion into and modification of a relation, and that may cause accidental loss of information during a deletion from a relation
- Waste of storage space due to NULLs and the difficulty of performing selections, aggregation operations, and joins due to NULL values
- Generation of invalid and spurious data during joins on base relations with matched attributes that may not represent a proper (foreign key, primary key) relationship

The strategy for achieving a good design is to decompose a badly designed relation appropriately.

4.4 Functional Dependencies

The single most important concept in relational schema design theory is that of a functional dependency.

Definition of Functional Dependency

A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has n attributes $A1, A2, ..., An$.
If we think of the whole database as being described by a single **universal** relation schema $R = \{A1, A2, ... , An\}$.
A **functional dependency**, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R, *such that* any two tuples $t1$ and $t2$ in r that have $t1[X] = t2[X]$, they must also have $t1[Y] = t2[Y]$.
This means that the values of the Y component of a tuple in r depend on, or are *determined by*, the values of the X component;
We say that the values of the X component of a tuple uniquely (or **functionally**) *determine* the values of the Y component.
We say that there is a functional dependency from X to Y, or that Y is **functionally dependent** on X.

Functional Dependency:

Functional dependency is represented as **FD** or **f.d.** The set of attributes X is called the **left-hand side** of the FD, and Y is called the **right-hand side**.
X functionally determines Y in a relation schema R if, and only if, whenever two tuples of $r(R)$ agree on their X-value, they must necessarily agree on their Y-value.

If a constraint on *R* states that there cannot be more than one tuple with a given *X*-value in any relation instance *r*(*R*)—that is, *X* is a **candidate key** of *R*— this implies that *X* -> *Y* for any subset of attributes *Y* of *R*.
If *X* is a candidate key of *R*, then *X*-> *R*.
If *X* -> *Y* in *R*, this does not imply that *Y* -> *X* in *R*.
A functional dependency is a property of the **semantics** or **meaning of the attributes**.
Whenever the semantics of two sets of attributes in *R* indicate that a functional dependency should hold, we specify the dependency as a constraint.
Legal Relation States:
Relation extensions *r*(*R*) that satisfy the functional dependency constraints are called **legal relation states** (or **legal extensions**) of *R*.
Functional dependencies are used to describe further a relation schema *R* by specifying constraints on its attributes that must hold *at all times.*
Certain FDs can be specified without referring to a specific relation, but as a property of those attributes given their commonly understood meaning.
For example, {State, Driver_license_number} -> Ssn should hold for any adult in the United States and hence should hold whenever these attributes appear in a relation.
Consider the relation schema EMP_PROJ from the semantics of the attributes and the relation, we know that the following functional dependencies should hold:
a. Ssn -> Ename
b. Pnumber -> {Pname, Plocation}
c. {Ssn, Pnumber}-> Hours

A functional dependency is a *property of the relation schema R*, not of a particular legal relation state *r* of *R*. Therefore, an FD *cannot* be inferred automatically from a given relation extension *r* but must be defined explicitly by someone who knows the semantics of the attributes of *R*.

4.5 CLOSURE OF A SET OF FUNCTIONAL DEPEDENCIES

Given a relational schema R, a functional dependencies f on R is logically implied by a set of functional dependencies F on R if every relation instance r(R) that satisfies F also satisfies f.

The closure of F, denoted by F^+, is the set of all functional dependencies logically implied by F.
The closure of F can be found by using a collection of rules called **Armstrong axioms.**
Reflexivity rule: If A *is a set of attributes and B is subset or equal to A, then A*→B holds.
Augmentation rule: If A→B holds and C is a set of attributes, then CA→CB holds
Transitivity rule: If A→B holds and B→C holds, then A→C holds.
Union rule: If A→B holds and A→C then A→BC holds
Decomposition rule: If A→BC holds, then A→B holds and A→C holds.
Pseudo transitivity rule: If A→B holds and BC→D holds, then AC→D holds.

Suppose we are given a relation schema R=(A,B,C,G,H,I) and the set of function dependencies
A→B,A→C,CG→H,CG→I,B→H
We list several members of F^+ here:

- A→H, since A→B and B→H hold, we apply the transitivity rule.
- CG→HI. Since CG→H and CG→I , the union rule implies that CG→HI
- AG→I, since A→C and CG→I, the pseudo transitivity rule implies that AG→I holds

Algorithm of compute F+ :
To compute the closure of a set of functional dependencies F:

F+ = F
repeat
for each functional dependency *f* in *F+*

apply reflexivity and augmentation rules on *f*
add the resulting functional dependencies to *F*+
for each pair of functional dependencies *f*1and *f*2 in *F*+
if *f*1 and *f*2 can be combined using transitivity
then add the resulting functional dependency to *F*+
until *F*+ does not change any further

Closure of Attribute sets:-

To test whether a set α is a super key, we must devise an algorithm for computing the set of attributes functionally determined by alpha. One way of doing this is to compute F+ take all functional dependencies. However doing so can be expensive, since F+ can be large.

Given a set of attributes α, define the *closure* of α under *F* (denoted by α^+) as the set of attributes that are functionally determined by α under *F:* $\alpha \rightarrow \beta$ is in F^+ <=> $\beta \subseteq \alpha^+$

Algorithm to compute α^+, the closure of α under *F* *result* := α;
while (changes to *result*) **do**
for each $\beta \rightarrow \gamma$ **in** *F* **do**
begin
if $\beta \subseteq$ *result* **then** *result* := *result* $\cup \gamma$
end

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:
Testing for superkey:
To test if α is a superkey, we compute α^+, and check if α^+ contains all attributes of *R*.
Testing functional dependencies
To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$. .
That is, we compute α^+ by using attribute closure, and then check if it contains β.
Is a simple and cheap test, and very useful
Computing closure of F
For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

4.6 Canonical Cover

Sets of functional dependencies may have redundant dependencies that can be inferred from the others
Eg: A -> C is redundant in: {A -> B, B -> C, A -> C}
Parts of a functional dependency may be redundant
E.g. on RHS: {A -> B, B -> C, A -> CD} can be simplified to
{A -> B, B -> C, A -> D}
E.g. on LHS: {A -> B, B -> C, AC -> D} can be simplified to
{A -> B, B -> C, A -> D}
Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

4.7 Dependency preservation

Getting lossless decomposition is necessary. But of course, we also want to keep dependencies, since losing a dependency means, that the corresponding constraint can be check only through natural join of the appropriate resultant relation in the decomposition. This would be very

expensive, so, our aim is to get a lossless dependency preserving decomposition.

Example:

R=(A, B, C), F={A -> B, B -> C}
Decomposition of R: R1=(A, C) R2=(B, C)
Does this decomposition preserve the given dependencies?

Solution:

In R1 the following dependencies hold: F1'={A -> A, C -> C, A -> C, AC -> AC}
In R2 the following dependencies hold: F2'= {B -> B, C -> C, B -> C, BC -> BC}
The set of nontrivial dependencies hold on R1 and R2: F':= {B -> C, A -> C}

A -> B cannot be derived from F', so this decomposition is NOT dependency preserving.

Example:

R=(A, B, C), F={A -> B, B -> C}
Decomposition of R: R1=(A, B) R2=(B, C)
Does this decomposition preserve the given dependencies?

Solution:

In R1 the following dependencies hold: F1={A -> B, A -> A, B -> B, AB -> AB}
In R1 the following dependencies hold: F2= {B -> B, C -> C, B -> C, BC -> BC}
F'= F1' U F2' = {A -> B, B -> C, trivial dependencies}

In F' all the original dependencies occur, so this decomposition preserves dependencies.

Full functional dependency:

Given a relational scheme R and an FD X→Y ,Y is fully functional dependent on X if there is no Z, where Z is a proper subset of X such that Z→Y. The dependency X→Y is left reduced, there being no extraneous attributes attributes in the left hand side of the dependency.

Partial dependency:

Given a relation dependencies F defined on the attributes of R and K as a candidate key ,if X is a proper subset of K and if F|= X→A, then A is said to be partial dependent on K

Prime attribute and non prime attribute:

A attribute A in a relation scheme R is a **prime attribute** or simply **prime** if A is part of any candidate key of the relation. If A is not a part of any candidate key of R, A is called a nonprime attribute or simply **non prime** .

Trivial functional dependency:

A FD X→Y is said to be a trivial functional dependency if Y is subset of X and X U Y = R, R {X, Y}is the relation.

4.8 Normal Forms Based on Primary Keys

Normalization of data is a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of
(1) minimizing redundancy and
(2) minimizing the insertion, deletion, and update anomalies.
It can be considered as a "filtering" or "purification" process to make the design have successively better quality.

We assume that a set of functional dependencies is given for each relation, and that each relation has a designated primary key.
Each relation is then evaluated for adequacy and decomposed further as needed to achieve higher normal forms, using the normalization theory.
We focus on the first three normal forms for relation schemas and the intuition behind them, and discuss how they were developed historically.
More general definitions of these normal forms, which take into account all candidate keys of a relation rather than just the primary key.

Normalization of Relations

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**.
The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as *relational design by analysis.* Initially, Codd proposed three normal forms, which he called first, second, and third normal form.
A stronger definition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation.
The normalization procedure provides database designers with: A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes.
A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree
Definition.
The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized.
Normal forms, when considered *in isolation* from other factors, do not guarantee a good database design. It is generally not sufficient to check separately that each relation schema in the database is in a given normal form.
Rather, the process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas, taken together, should possess. These would include two properties:
The **nonadditive join or lossless join property**, which guarantees that the spurious tuple generation problem does not occur with respect to the relation schemas created after decomposition.
The **dependency preservation property**, which ensures that each functional dependency is represented in some individual relation resulting after decomposition. The nonadditive join property is extremely critical and **must be achieved at any cost**.
Practical Use of Normal Forms
Most practical design projects acquire existing designs of databases from previous designs, designs in legacy models, or from existing files.
Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties stated previously.
Although several higher normal forms have been defined, database design as practiced in industry today pays particular attention to normalization only up to 3NF, BCNF, or at most 4NF.

Another point worth noting is that the database designers *need not* normalize to the highest possible normal form. Relations may be left in a lower normalization status, such as 2NF, for performance reason.
Denormalization is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form.

4.9 Definitions of Keys and Attributes Participating in Keys

Definition: A **superkey** of a relation schema $R = \{A1, A2, \ldots, An\}$ is a set of attributes $S \subseteq R$ with the property that no two tuples $t1$ and $t2$ in any legal relation state r of R will have $t1[S] = t2[S]$.

A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey anymore. The difference between a key and a superkey is that a key has to be *minimal;* that is, if we have a key
$K = \{A1, A2, ..., Ak\}$ of R, then $K - \{Ai\}$ is not a key of R for any Ai, $1 \leq i \leq k$
{Ssn} is a key for EMPLOYEE, whereas {Ssn}, {Ssn, Ename},

{Ssn, Ename, Bdate}, and any set of attributes that includes Ssn are all superkeys.
If a relation schema has more than one key, each is called a **candidate key**.
One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called secondary keys.
In a practical relational database, each relation schema must have a primary key. If no candidate key is known for a relation, the entire relation can be treated as a default superkey. In the Table EMPLOYEE, {Ssn} is the only candidate key for EMPLOYEE, so it is also the primary key.
Definition:
An attribute of relation schema *R* is called a **prime attribute** of *R* if it is a member of *some candidate key* of *R.*
An attribute is called **nonprime** if it is not a prime attribute—that is, if it is not a member of any candidate key, both Ssn and Pnumber are prime attributes of WORKS_ON, whereas other attributes of WORKS_ON are nonprime.
We now present the first three normal forms: 1NF, 2NF, and 3NF.
As we shall see, 2NF and 3NF attack different problems.

4.10 First Normal Form

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model.
It states that:
1. the domain of an attribute must include only *atomic* (simple, indivisible) *values* and
2. that the value of any attribute in a tuple must be a *single value* from the domain of that attribute.
Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple.* In other words, 1NF disallows *relations within relations* or *relations as attribute values within tuples*.
The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.
Consider the DEPARTMENT relation schema, whose primary key is Dnumber, and suppose that we extend it by including the Dlocations attribute.
We assume that each department can have *a number of* locations.
As we can see, this is not in 1NF because Dlocations is not an atomic attribute. There are two ways we look at the Dlocations attribute:

The domain of Dlocations contains atomic values, but some tuples can have a set of these values. In this case, Dlocations is not functionally dependent on the primary key Dnum.
First normal form also disallows multi-valued attributes that are themselves composite. These are called **nested relations** because each tuple can have a relation *within it.*
This procedure can be applied recursively to a relation with multiple-level nesting to **unnest** the relation into a set of 1NF relations. This is useful in converting an unnormalized relation schema with many levels of nesting into 1NF relations.

4.11 Second Normal Form

Second normal form (2NF) is based on the concept of *full functional dependency.*
Functional Dependency:
The attribute B is fully functionally dependent on the attribute A if each value of A determines one and only one value of B.
Example: PROJ_NUM-> PROJ_NAME
In this case, the attribute PROJ_NUM is known as the determinant attribute and the attribute PROJ_NAME is known as the dependent attribute.
Generalized Definition:
Attribute A determines attribute B (that is B is functionally dependent on A) if all of the rows in the table that agree in value for attribute A also agree in value for attribute B.
Fully functional dependency (composite key)

If attribute B is functionally dependent on a composite key A but not on any subset of that composite key, the attribute B is fully functionally dependent on A.
Partial Dependency:
When there is a functional dependence in which the determinant is only part of the primary key, then there is a partial dependency.
For example if (A, B) -> (C, D) and B -> C and (A, B) is the primary key, then the functional dependence B -> C is a partial dependency.
{Ssn, Pnumber} -> Hours is a full dependency (neither Ssn -> Hours nor Pnumber -> Hours holds).
However, the dependency {Ssn, Pnumber}-> Ename is partial because Ssn -> Ename holds.
Transitive Dependency:
When there are the following functional dependencies such that X -> Y, Y -> Z and X is the primary key, then X -> Z is a transitive dependency because X determines the value of Z via Y. Whenever a functional dependency is detected amongst nonprime, there is a transitive dependency.

Definition. A relation schema *R* is in 2NF if every nonprime attribute *A* in *R* is *fully functionally dependent* on the primary key of *R*.
The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key.
If the primary key contains a single attribute, the test need not be applied at all.
If a relation schema is not in 2NF, it can be *second normalized* or *2NF normalized* into a number of 2NF relations in which nonprime attributes are associated only with the part of the primary key on which they are fully functionally dependent.

4.12 Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*.
A functional dependency *X -> Y* in a relation schema *R* is a **transitive dependency** if there exists a set of attributes *Z* in *R* that is neither a candidate key nor a subset of any key of *R*, and both *X -> Z* and *Z -> Y* hold.
Definition. According to Codd's original definition, a relation schema *R* is in **3NF** if it satisfies 2NF *and* no nonprime attribute of *R* is transitively dependent on the primary key.

General Definitions of Second and Third Normal Forms
In general, we want to design our relation schemas so that they have neither partial nor transitive dependencies because these types of dependencies cause the update anomalies seen previously.
The steps for normalization into 3NF relations that we have discussed so far disallow partial and transitive dependencies on the *primary key.* The normalization procedure described so far is useful for analysis in practical situations for a given database where primary keys have already been defined.
As a general definition of **prime attribute**, an attribute that is part of *any candidate key* will be considered as prime. Partial and full functional dependencies and transitive dependencies will now be considered *with respect to all candidate keys* of a relation. Prime attributes are part of any candidate key Non-prime attribute are not.

General Definition of Second Normal Form

A relation schema *R* is in **second normal form (2NF)** if every nonprime attribute *A* in *R* is not partially dependent on *any* key of *R*.
The test for 2NF involves testing for functional dependencies whose left-hand side attributes are *part of* the primary key. If the primary key contains a single attribute, the test need not be applied at all.
General Definition of Third Normal Form
A relation schema *R* is in **third normal form (3NF)** if, whenever a *nontrivial* functional dependency *X -> A* holds in *R*, either
(a) *X* is a superkey of *R*, or

(b) *A* is a prime attribute of *R*.

Interpreting the General Definition of Third Normal Form

A relation schema *R* violates the general definition of 3NF if a functional dependency *X -> A* holds in *R* that does not meet either condition—meaning that it violates *both* conditions (a) and (b) of 3NF. This can occur due to two types of problematic functional dependencies:

A nonprime attribute determines another nonprime attribute. Here we typically have a transitive dependency that violates 3NF.

A proper subset of a key of *R* functionally determines a nonprime attribute. Here we have a partial dependency that violates 3NF (and also 2NF).

Therefore, we can state a **general alternative definition of 3NF** as follows:

Alternative Definition. A relation schema *R* is in 3NF if every nonprime attribute of *R* meets both of the following conditions:

- It is fully functionally dependent on every key of *R*.
- It is nontransitively dependent on every key of *R*.

4.13 Boyce-Codd Normal Form

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF.

Definition: A relation schema *R* is in **BCNF** if whenever a nontrivial functional dependency X -> A holds in R, then X is a superkey of R. In practice, most relation schemas that are in 3NF are also in BCNF.

Only if *X -> A* holds in a relation schema *R* with *X* not being a superkey *and A* being a prime attribute will *R* be in 3NF but not in BCNF. Ideally, relational database design should strive to achieve BCNF or 3NF for every relation schema.

4.14 Multivalued Dependency and Fourth Normal Form

- **Multivalued dependency (MVD)**

  A multivalued dependency X ->> Y specified on relation schema R, where X and Y are both subsets of R, species the following constraint on any relation state r of R; If two tuples t1 and t2 exist in r with the following properties, where we use Z to denote (R – (X U Y)):

  t3[X] = t4 [X] = t1[X] = t2[X]

  t3[Y] = t1 [Y] and t4[Y] = t2[Y]

  t3[Z] = t2 [Z] and t4[Z] = t1[Z]

  Relations containing nontrivial MVDs

  All-key relations

  Fourth Normal Form (4NF)

  Violated when a relation has undesirable multivalued dependencies.

  Definition of 4NF:

  A relation schema R is in 4NF with respect to a set of dependencies F (that includes functional dependencies and multivalued dependencies) if, for every nontrivial multivalued dependency X ->> Y in F^+ , X is a super key for R.

4.15 Join dependency and 5NF

- Multiway decomposition into fifth normal form (5NF)
- Very peculiar semantic constraint
- Normalization into 5NF is very rarely done in practice.

  Join Dependencies:

  A join dependency (JD), denoted by JD (R1, R2, ...,Rn), specified on relation schema R, species a constraint on the states r of R. The constraint states that every legal state r of R

should have a nonadditive join decomposition into R1, R2, ...Rn. Hence, for every such r we have

- $(\prod_{R1}(r), \prod_{R2}(r),\ldots\ldots, \prod_{Rn}(r)) = r$

  Fifth Normal Form (5NF):

  A relation schema R is in 5NF or Project-join normal form (PJNF) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD (R1, R2, ... Rn) in F^+ (that is, implied by F), every Ri is a superkey of R.

4.16 Lossless and Lossy decomposition:

To Identify whether a decomposition is lossless, it must satisfy the following conditions :

1. $R_1 \cup R_2 = R$
2. $R_1 \cap R_2 \neq \Phi$ and
3. $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$

1) Case 1 :

R(ABC)
F = {A → B, A → C} decomposed into
D = R_1(AB), R_2(BC)
Find whether D is Lossless or Lossy ?
Solution :
D = {AB, BC}
Step 1: AB ∪ BC = ABC
Step 2: AB ∩ BC = B //Intersection
Step 3: B^+ = {B} //Not a super key of R_1 or R_2
⇒ Decomposition is lossy.

2) Case 2 :

R(ABCDEF)
F = {A → B, B → C, C → D, E → F} decomposed into
D = R_1(AB), R_2(BCD), R_3(DEF).
Find whether D is Lossless or Lossy ?
Solution :
Step 1: AB ∪ BCD ∪ DEF = ABCDEF = R // Condition 1 satisfies
step 2: AB ∩ BCD = B
B^+ = {BCD} //superkey of R_2
⇒ R_{12}(ABCD)

ABCD ∩ DEF = D
D^+ = {D} // Not a superkey of R_{12} or R_3
⇒ Decomposition is Lossy.

3) Case 3:

R(ABCDEG)
F = {AB → C, AC → B, AD → E, B → D, BC → A, E → G} decomposed into
(i) D1 = R_1(AB), R_2(BC), R_3(ABDE), R_4(EG).
(ii) D2 = R_1(ABC), R_2(ACDE), R_3(ADG).
Find whether D1 and D2 is Lossless or Lossy ?

Solution (i) :

Step 1: AB ∪ BC ∪ ABDE ∪ EG = ABCDEG = R // Condition 1 satisfies

step 2: AB ∩ BC = B

B^+ = {BD} //Not a superkey of R1 or R_2

⇒ Decomposition is Lossy. No need to check further.

Solution (ii) :

Step 1: ABC ∪ ACDE ∪ ADG = ABCDEG = R // Condition 1 satisfies

step 2: ABC ∩ ACDE = AC

AC^+ = {ACBDEG} //superkey

⇒ R_{12}(ABCDE)

ABCDE ∩ ADG = AD

AD^+ = {ADEG} //Superkey of R_3

⇒ R_{123}(ABCDEG)

⇒ Decomposition is LossLess.

MODULE 4 MCQ and Short type Problem

Multiple choice questions (MCQ)

1. In the _____ normal form, a composite attribute is converted to individual attributes.
   A) First
   B) Second
   C) Third
   D) Fourth

   Ans. (A)

2. A table on the many side of a one to many or many to many relationship must:
   a) Be in Second Normal Form (2NF)
   b) Be in Third Normal Form (3NF)
   c) Have a single attribute key
   d) Have a composite key

   Ans. (d)

3. Tables in second normal form (2NF):
   a) Eliminate all hidden dependencies
   b) Eliminate the possibility of a insertion anomalies
   c) Have a composite key
   d) Have all non key fields depend on the whole primary key
   Ans. (d)
4. Which-one of the following statements about normal forms is FALSE?
   a) BCNF is stricter than 3 NF
   b) Lossless, dependency -preserving decomposition into 3 NF is always possible
   c) Loss less, dependency – preserving decomposition into BCNF is always possible
   d) Any relation with two attributes is BCNF
   Ans. (d)

Short question

1) Explain why BCNF is stricter than 3NF.
2) What is FD, MVD, JD?
3) What are the advantages of normalization?

Broad question

1 What are different anomalies in database design? Explain each type with examples.

2. What is Normalization? Explain 3NF with example. Why BCNF is considered to be stronger Normal form than 3NF?

3. What is multivalued dependency? Explain 4NF and 5NF with proper examples.

Text Books:

1. Henry F. Korth and Silberschatz Abraham, "Database System Concepts", Mc.Graw Hill.
2. Elmasri Ramez and Novathe Shamkant, "Fundamentals of Database Systems", Benjamin Cummings Publishing. Company.
3. Date C. J., "Introduction to Database Management", Vol. I, II, III, Addison Wesley.

Module 5:
Internals of RDBMS [9L]

Physical data structures, Query optimization: join algorithm, statistics and cost based optimization. Transaction processing, Concurrency control and Recovery Management: transaction model properties, state serializability, lock base protocols; two phase locking, Dead Lock handling

5.1 Physical Data Structures:

A physical data model (or database design) is a representation of a data design as implemented, or intended to be implemented, in a database management system. In the lifecycle of a project it typically derives from a logical data model, though it may be reverse-engineered from a given database implementation. A complete physical data model will include all the database artifacts required to create relationships between tables or to achieve performance goals, such as indexes, constraint definitions, linking tables, partitioned tables or clusters. Analysts can usually use a physical data model to calculate storage estimates; it may include specific storage allocation details for a given database system.

As of 2012 seven main databases dominate the commercial marketplace: Informix, Oracle, Postgres, SQL Server, Sybase, DB2 and MySQL. Other RDBMS systems tend either to be legacy databases or used within academia such as universities or further education colleges. Physical data models for each implementation would differ significantly, not least due to underlying operating-system requirements that may sit underneath them. For example: SQL Server runs only on Microsoft Windows operating-systems, while Oracle and MySQL can run on Solaris, Linux and other UNIX-based operating-systems as well as on Windows. This means that the disk requirements, security requirements and many other aspects of a physical data model

will be influenced by the RDBMS that a database administrator (or an organization) chooses to use.

Data are actually stored as bits, or numbers and strings, but it is difficult to work with data at this level.

It is necessary to view data at different levels of abstraction.

Schema:

- Description of data at some level. Each level has its own schema.

We will be concerned with three forms of schemas:

- physical,
- conceptual, and
- external.

Physical Data Level

The **physical schema** describes details of how data is stored: files, indices, etc. on the random access disk system. It also typically describes the record layout of files and type of files (hash, b-tree, flat).

Early applications worked at this level - explicitly dealt with details. E.g., minimizing physical distances between related data and organizing the data structures within the file (blocked records, linked lists of blocks, etc.)

Problem:

- Routines are hardcoded to deal with physical representation.
- Changes to data structures are difficult to make.
- Application code becomes complex since it must deal with details.
- Rapid implementation of new features very difficult.

Conceptual Data Level

Also referred to as the Logical level

Hides details of the physical level.

- In the relational model, the conceptual schema presents data as a set of tables.

The DBMS maps data access between the conceptual to physical schemas automatically.

- Physical schema can be changed without changing application:
- DBMS must change mapping from conceptual to physical.
- Referred to as **physical data independence**.

External Data Level

In the relational model, the **external schema** also presents data as a set of relations. An external schema specifies a **view** of the data in terms of the conceptual level. It is tailored to the needs of

a particular category of users. Portions of stored data should not be seen by some users and begins to implement a level of security and simplifies the view for these users

Examples:

- Students should not see faculty salaries.
- Faculty should not see billing or payment data.

Information that can be derived from stored data might be viewed as if it were stored.

- GPA not stored, calculated when needed.

Applications are written in terms of an external schema. The external view is computed when accessed. It is not stored. Different external schemas can be provided to different categories of users. Translation from external level to conceptual level is done automatically by DBMS at run time. The conceptual schema can be changed without changing application:

- Mapping from external to conceptual must be changed.
- Referred to as **conceptual data independence**.

RELATIONAL CONSTRAINTS:

There are three types of constraints on relational database that include

- DOMAIN CONSTRAINTS
- KEY CONSTRAINTS
- INTEGRITY CONSTRAINTS

DOMAIN CONSTRAINTS:

It specifies that each attribute in a relation an atomic value from the corresponding domains. The data types associated with commercial RDBMS domains include:

- Standard numeric data types for integer
- Real numbers
- Characters
- Fixed length strings and variable length strings

Thus, domain constraints specifies the condition that we to put on each instance of the relation. So the values that appear in each column must be drawn from the domain associated with that column as in table 5.1.

Table 5.1: Student table

Rollno	Name	City	Age
101	Sujit	Bam	23
102	kunal	bbsr	22

Key constraints:

This constraints states that the key attribute value in each tuple msut be unique .i.e, no two tuples contain the same value for the key attribute.(null values can allowed)

Emp(empcode,name,address) . here empcode can be unique

Integrity constraints:

There are two types of integrity constraints:

- Entity integrity constraints
- Referential integrity constraints

Entity integrity constraints:
It states that no primary key value can be null and unique. This is because the primary key is used to identify individual tuple in the relation. So we will not be able to identify the records uniquely containing null values for the primary key attributes. This constraint is specified on one individual relation.
Referential integrity constraints:
It states that the tuple in one relation that refers to another relation must refer to an existing tuple in that relation. This constraints is specified on two relations .
If a column is declared as foreign key that must be primary key of another table.
Department(deptcode,dname)
Here the deptcode is the primary key.
Emp(*empcode*,name,city,*deptcode*).
Here the deptcode is foreign key.

CODD'S RULES

Rule 1 : The information Rule.

"All information in a relational data base is represented explicitly at the logical level and in exactly one way - by values in tables."

Everything within the database exists in tables and is accessed via table access routines.

Rule 2 : Guaranteed access Rule.

"Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value and column name."

To access any data-item you specify which column within which table it exists, there is no reading of characters 10 to 20 of a 255 byte string.

Rule 3 : Systematic treatment of null values.

"Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type."

If data does not exist or does not apply then a value of NULL is applied, this is understood by the RDBMS as meaning non-applicable data.

Rule 4 : Dynamic on-line catalog based on the relational model.

"The data base description is represented at the logical level in the same way as-ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data."

The Data Dictionary is held within the RDBMS, thus there is no-need for off-line volumes to tell you the structure of the database.

Rule 5 : Comprehensive data sub-language Rule.

"A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all the following items

4. Data Definition
5. View Definition
6. Data Manipulation (Interactive and by program).
7. Integrity Constraints
8. Authorization.

Every RDBMS should provide a language to allow the user to query the contents of the RDBMS and also manipulate the contents of the RDBMS.

Rule 6 : .View updating Rule

"All views that are theoretically updateable are also updateable by the system."

Not only can the user modify data, but so can the RDBMS when the user is not logged-in.

Rule 7 : High-level insert, update and delete.

"The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update and deletion of data."

The user should be able to modify several tables by modifying the view to which they act as base tables.

Rule 8 : Physical data independence.

"Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods."

The user should not be aware of where or upon which media data-files are stored

Rule 9 : Logical data independence.

"Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables."

User programs and the user should not be aware of any changes to the structure of the tables (such as the addition of extra columns).

Rule 10 : Integrity independence.

"Integrity constraints specific to a particular relational data base must be definable in the relational data sub-language and storable in the catalog, not in the application programs."

If a column only accepts certain values, then it is the RDBMS which enforces these constraints and not the user program, this means that an invalid value can never be entered into this column, whilst if the constraints were enforced via programs there is always a chance that a buggy program might allow incorrect values into the system.

Rule 11 : Distribution independence.

"A relational DBMS has distribution independence."

The RDBMS may spread across more than one system and across several networks, however to the end-user the tables should appear no different to those that are local.

Rule 12 : Non-subversion Rule.

"If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity Rules and constraints expressed in the higher level relational language (multiple-records-at-a-time)."

5.2 Query Processing

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation
4. Execution

Parsing and translation

- translate the query into its internal form. This is then translated into relational algebra.
- Parser checks syntax, verifies relations

Evaluation

- The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

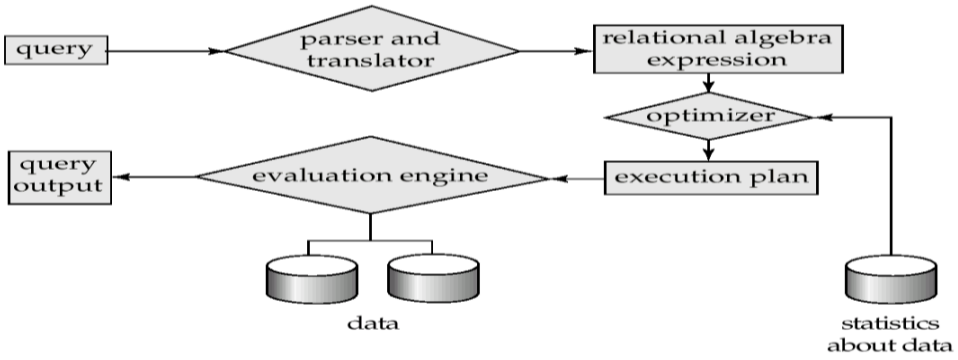


Figure. 5.1: Query processing

Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  E.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$ is equivalent to
  $\Pi_{balance}(\sigma_{balance<2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  E.g., can use an index on *balance* to find accounts with balance < 2500,
  or can perform complete relation scan and discard accounts with balance >= 2500

5.3 Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost.

Cost is estimated using statistical information from the database catalog e.g. number of tuples in each relation, size of tuples, etc.

- Here we study

a. How to measure query costs
b. Algorithms for evaluating relational algebra operations
c. How to combine algorithms for individual operations in order to evaluate a complete expression
d. How to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

5.4 Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU,* or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost

Cost to write a block is greater than cost to read a block

Data is read back after being written to ensure that the write was successful

- For simplicity we just use *number of block transfers from disk* as the cost measure
  - ★ We ignore the difference in cost between sequential and random I/O for simplicity
  - ★ We also ignore CPU costs for simplicity
- Costs depends on the size of the buffer in main memory
  - ★ Having more memory reduces need for disk access
  - ★ Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine ahead of actual execution
  - ★ We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Real systems take CPU cost into account, differentiate between sequential and random I/O, and take buffer size into account
- We do not include cost to writing output to disk in our cost formulae

5.4.1 Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfil a selection condition.
- Algorithm **A1** (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.
  - ★ Cost estimate (number of disk blocks scanned) = b_r
    - b_r denotes number of blocks containing records from relation *r*
  - ★ If selection is on a key attribute, cost = (b_r /2)
    - stop on finding record
  - ★ Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices
- **A2** *(binary search).* Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - ★ Assume that the blocks of a relation are stored contiguously
  - ★ Cost estimate (number of disk blocks to be scanned):
    - $\lceil \log_2(b_r) \rceil$ — cost of locating the first tuple by a binary search on the blocks
    - *Plus* number of blocks containing records that satisfy selection condition

5.4.2 Selections Using Indices

- **Index scan** – search algorithms that use an index
  - ★ Selection condition must be on search-key of index.

- **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
    - ★ *Cost* = HT_i + 1
- **A4** (*primary index on nonkey, equality)* Retrieve multiple records.
    - ★ Records will be on consecutive blocks
    - ★ *Cost* = HT_i + number of blocks containing retrieved records
- **A5** (*equality on search-key of secondary index).*
    - ★ Retrieve a single record if the search-key is a candidate key
        - *Cost* = HT_i + 1
    - ★ Retrieve multiple records if search-key is not a candidate key
        - Cost = HT_i + *number of records retrieved*
            - Can be very expensive!
        - each record may be on a different block
            - one block access for each retrieved record

5.4.3 Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$ by using
    - ★ a linear file scan or binary search,
    - ★ or by using indices in the following ways:
- **A6** (*primary index, comparison).* (Relation is sorted on A)

- For $\sigma_{A \ge V}(r)$ use index to find first tuple ≥ *v* and scan relation sequentially from there
- For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple > *v;* do not use index

- **A7** (secondary index, comparison).

- For $\sigma_{A \ge V}(r)$ use index to find first index entry ≥ v and scan index sequentially from there, to find pointers to records.
- For $\sigma_{A \le V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > *v*
    - In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper if many records are to be fetched!

5.4.4 Implementation of Complex Selections

- Implementation of Complex Selections Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
- **A8 (*conjunctive selection using one index).***
    - ★ Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}(r)$.
    - ★ Test other conditions on tuple after fetching it into memory buffer.
- A9 (conjunctive selection using multiple-key index).
    - ★ Use appropriate composite (multiple-key) index if available.
- A10 (conjunctive selection by intersection of identifiers).
    - ★ Requires indices with record pointers.
    - ★ Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
    - ★ Then fetch records from file
    - ★ If some conditions do not have appropriate indices, apply test in memory.

5.4.5 Algorithms for Complex Selections

- Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.

A11 (disjunctive selection by union of identifiers).

Applicable if all conditions have available indices.

Otherwise use linear scan.

Use corresponding index for each condition, and take union of all the obtained sets of record pointers.

Then fetch records from file

- **Negation:** $\sigma_{\neg\theta}(r)$
  - ★ Use linear scan on file
  - ★ If very few records satisfy $\neg\theta$, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

Query Optimization

5.4.6 Cost-Based Optimization

Consider finding the best join-order for $r_1 \quad r_2 \quad \ldots r_n$.

There are (2(n – 1))!/(n – 1)! different join orders for above expression. With n = 7, the number is 665280, with n = 10, the number is greater than 176 billion!

No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

5.4.7 Heuristic Optimization

Cost-based optimization is expensive, even with dynamic programming.

Systems may use heuristics to reduce the number of choices that must be made in a cost-based fashion.

Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:

- Perform selection early (reduces the number of tuples)
- Perform projection early (reduces the number of attributes)
- Perform most restrictive selection and join operations before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.
-

5.5 Query Tree Optimization Example•

What are the names of customers living on Elm Street who have checked out "Terminator"?•SQL query:

SELECT Name **FROM** Customer CU, CheckedOut CH,Film F **WHERE** Title= 'Terminator'**AND** F.FilmId = CH.FilmID **AND** CU.CustomerID = CH.CustomerID **AND** CU.Street = 'Elm'

Canonical query tree

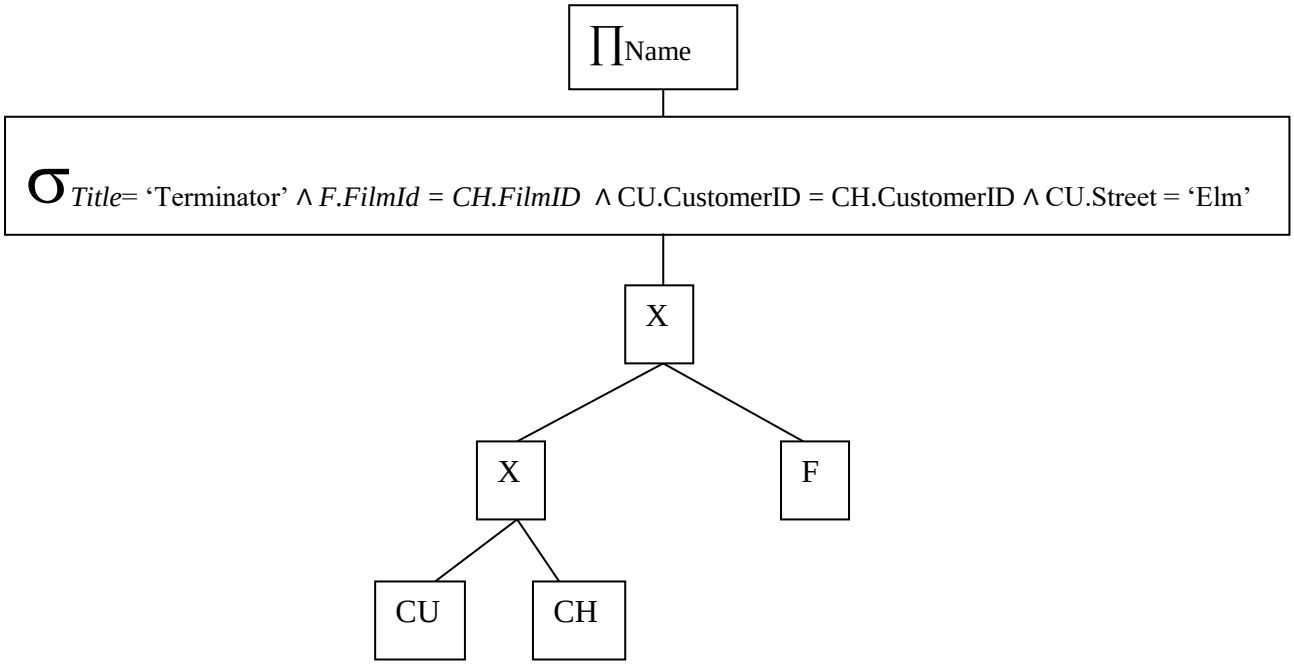


Figure 5.2: Query using Cartesian product

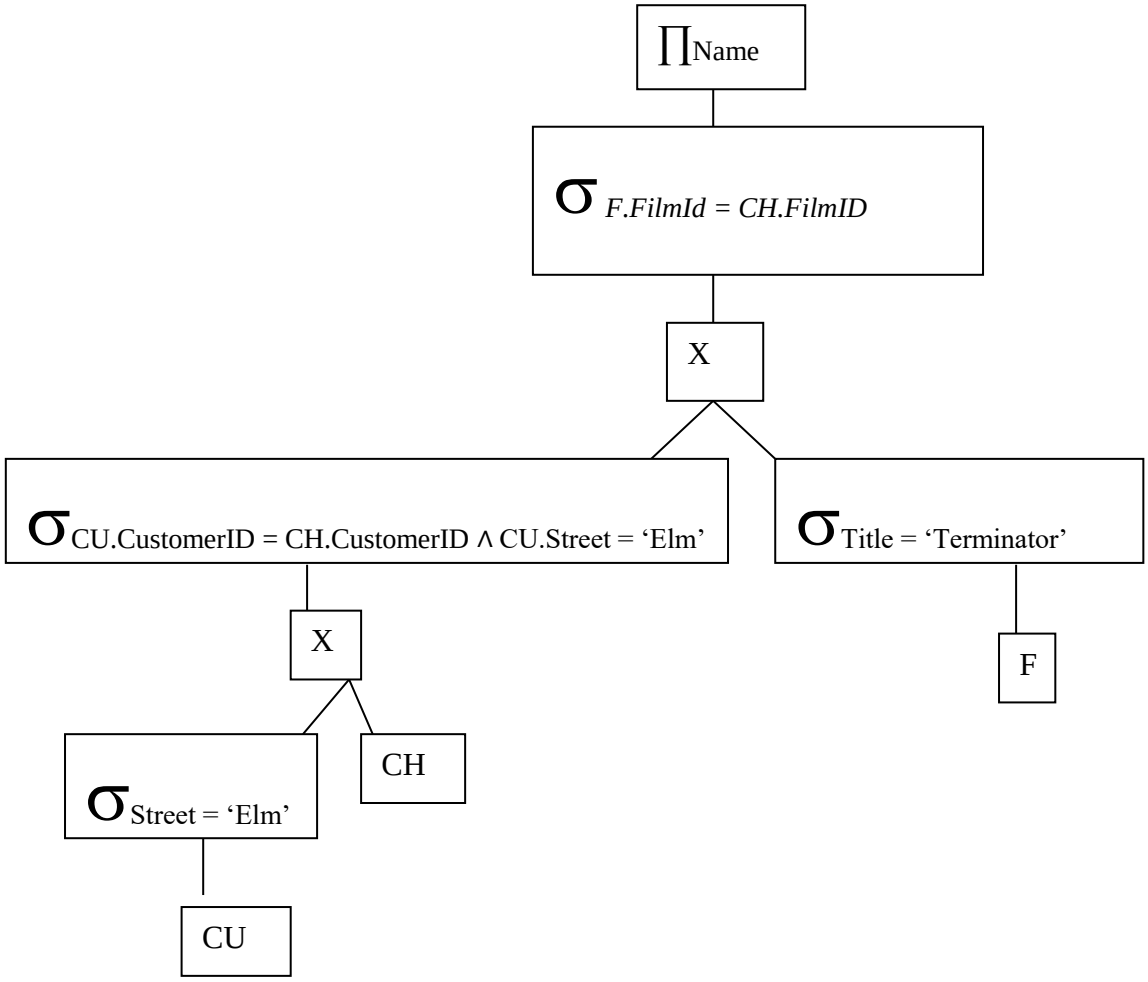


Figure 5.3: Query using applying selection early

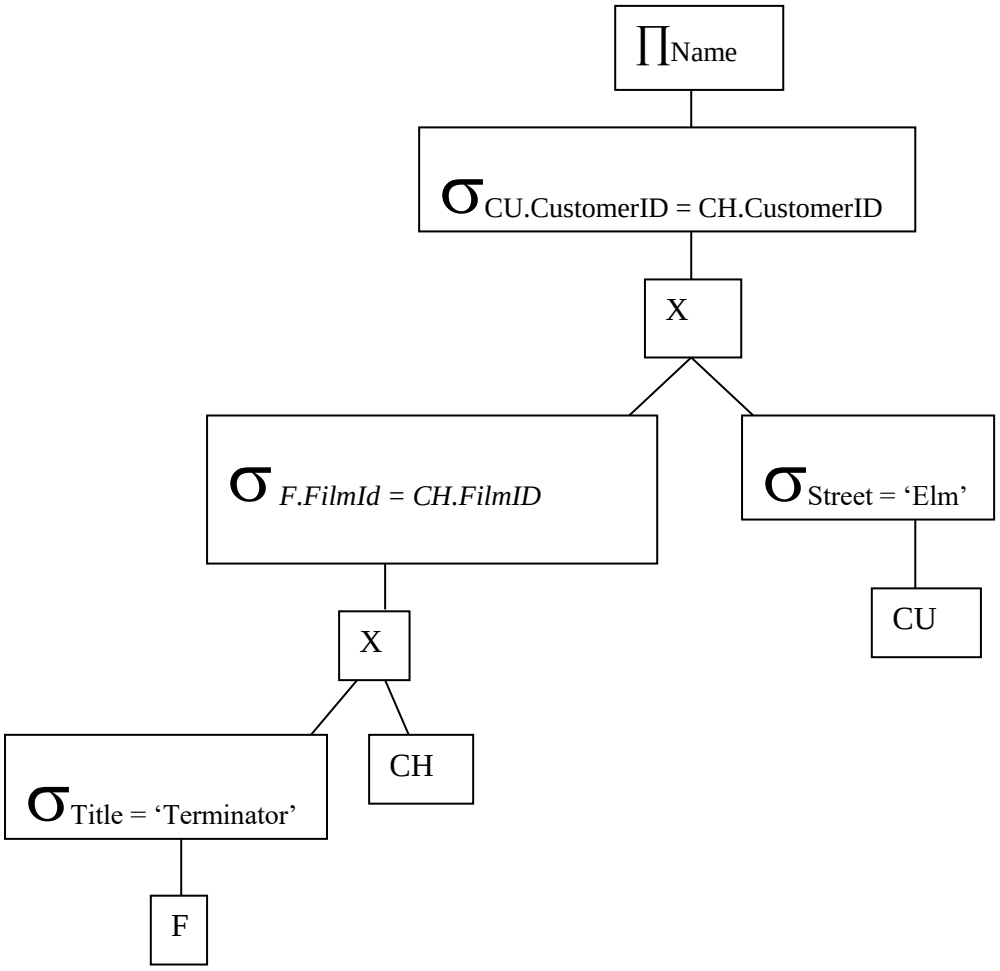


Figure 5.4: Query using applying more restrictive selections early

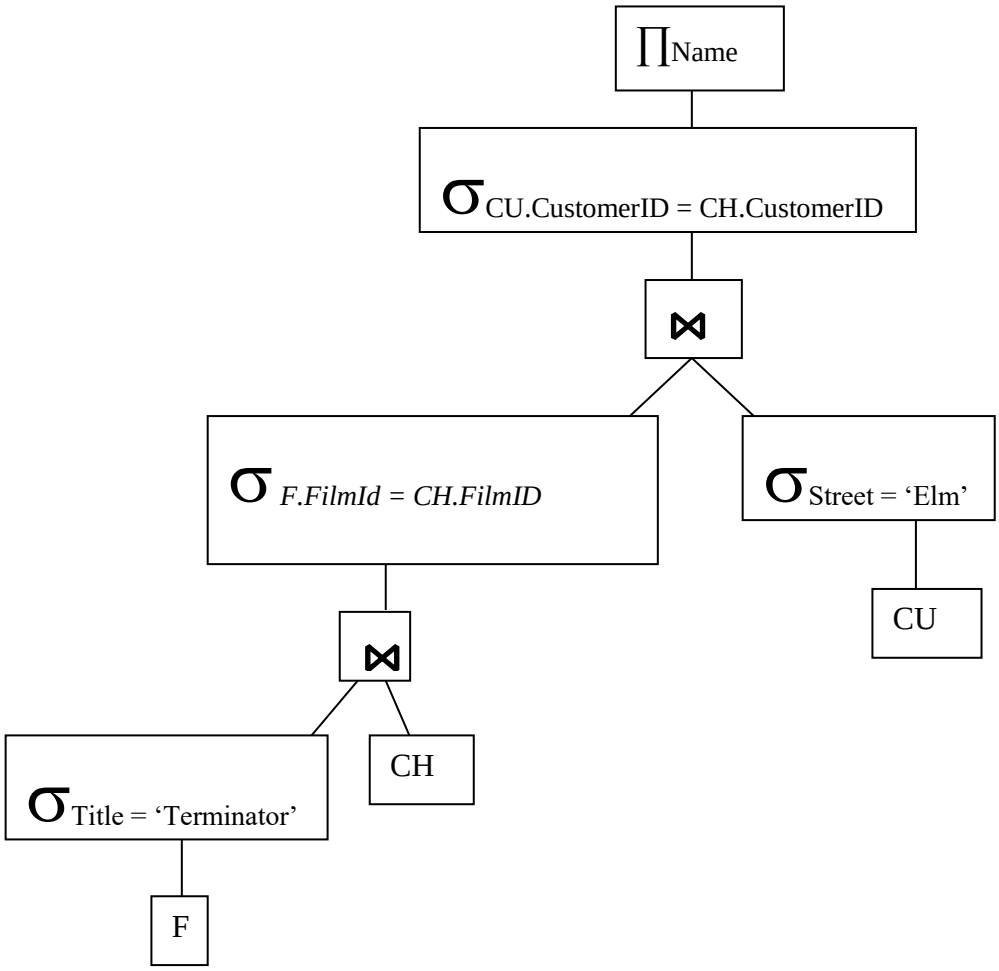


Figure 5.5: Query using forming joins

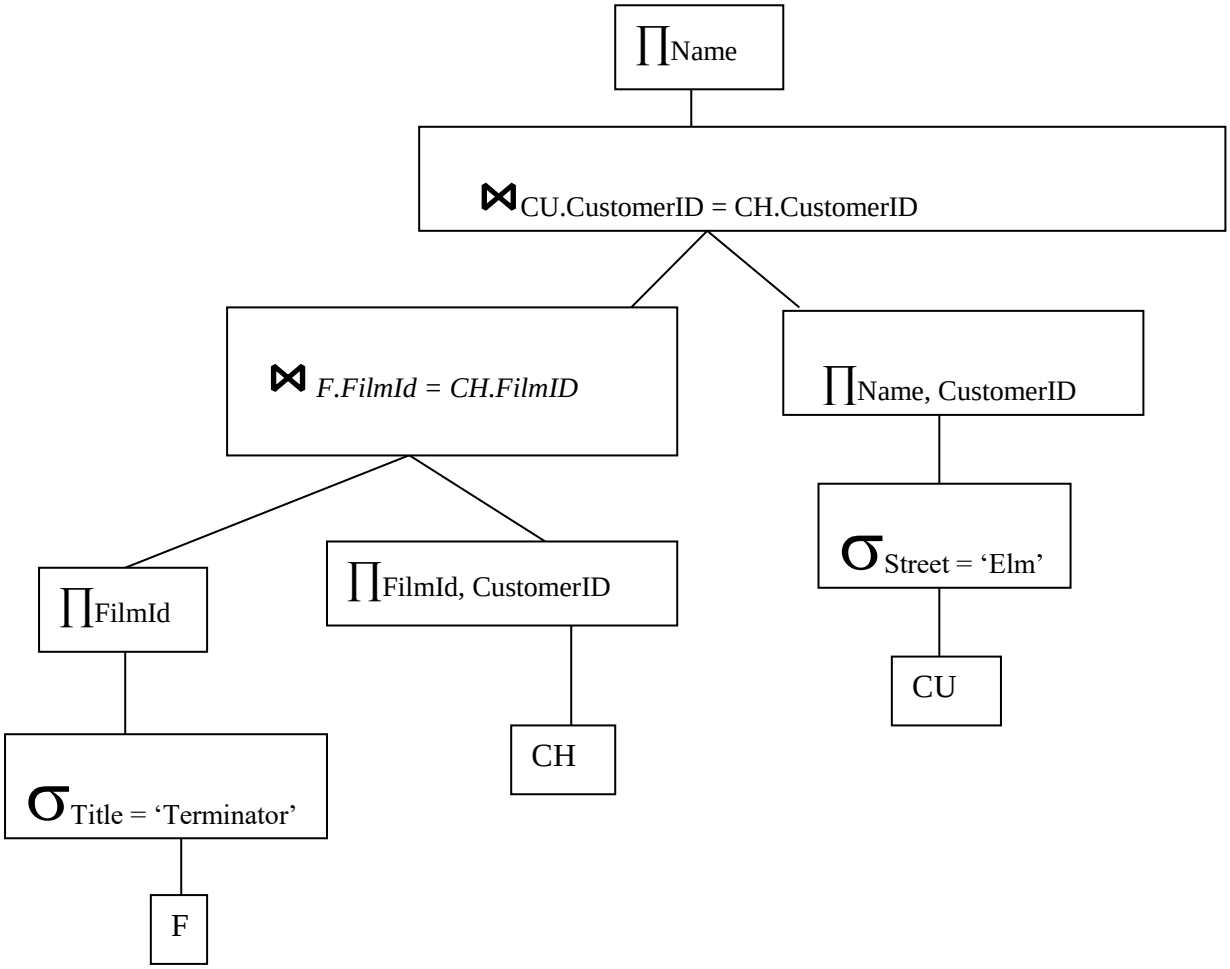


Figure 5.6: Query using applying projection early

5.5.1 Cost-Based Optimization

•Use transformations to generate multiple candidate query trees from the canonical query tree.
•Statistics on the inputs to each operator are needed.
- Statistics on leaf relations are stored in the system catalog.
- Statistics on intermediate relations must be estimated; most important is the relations' cardinalities.

•Cost formulas estimate the cost of executing each operation in each candidate query tree.
- Parameterized by statistics of the input relations.
- Also dependent on the specific algorithm used by the operator.
- Cost can be CPU time, I/O time, communication time, main memory usage, or a combination.

•The candidate query tree with the least total cost is selected for execution.

5.5.2 Relevant Statistics – depends on

- Per relation

- Tuple size
- Number of tuples (records): *r*
- Load factor (fill factor), percentage of space used in each block

Blocking factor (number of records per block): *bfr*
Relation size in blocks: *b*
Relation organization
Number of overflow blocks

- Per attribute

Attribute size and type
Number of distinct values for attribute A: d_A
Probability distribution over the values
Representation, e.g., compressed
Selection cardinality specifies the average size of
σA = *a*(*R*) for an arbitrary value *a*. (s*A*)
Could be maintained for the "average" attribute value, or on a per-value basis, as a histogram.

5.6 Introduction to Transaction Processing

5.6.1 Transaction: An executing program (process) that includes one or more database access operations

- Read operations (database retrieval, such as SQL SELECT)
- Write operations (modify database, such as SQL INSERT, UPDATE, DELETE)
- Transaction: A logical unit of database processing
- Example: Bank balance transfer of $100 dollars from a checking account to a saving account in a BANK database

Each execution of a program is a *distinct transaction* with different parameters

- Bank transfer program parameters: savings account number, checking account number, transfer amount

A transaction (set of operations) may be:

- stand-alone, specified in a high level language like SQL submitted interactively, or
- consist of database operations embedded within a program (most transactions)

Transaction boundaries: Begin and End transaction.

- An application program may contain several transactions separated by Begin and End transaction boundaries

Transaction Processing Systems: Large multi-user database systems supporting thousands of *concurrent transactions* (user processes) per minute

- Two Modes of Concurrency
  - Interleaved processing: concurrent execution of processes is interleaved in a single CPU
  - Parallel processing: processes are concurrently executed in multiple CPUs
  - Basic transaction processing theory assumes interleaved concurrency

For transaction processing purposes, a simple database model is used:

- **A database -** collection of named data items
- **Granularity (size) of a data item** - a field (data item value), a record, or a whole disk block
  - TP concepts are independent of granularity
- Basic operations on an item X:
  - **read_item(X)**: Reads a database item named X into a program variable. To simplify our notation, we assume that *the program variable is also named X.*
  - **write_item(X)**: Writes the value of program variable X into the database item named X.

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is one disk block (or page). A data item X (what is read or written) will usually be the field of some record in the database, although it may be a larger unit such as a whole record or even a whole block.
- **read_item(X) command includes the following steps:**
- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
- Copy item X from the buffer to the program variable named X.
- **write_item(X) command includes the following steps:**
- Find the address of the disk block that contains item X.
- Copy that disk block into a buffer in main memory (if it is not already in some main memory buffer).
- Copy item X from the program variable named X into its correct location in the buffer.
- Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Figure 5.7 shows two examples of transactions

- Notation focuses on the read and write operations
- Can also write in shorthand notation:
  - T1: b1; r1(X); w1(X); r1(Y); w1(Y); e1;
  - T2: b2; r2(Y); w2(Y); e2;
- bi and ei specify transaction boundaries (begin and end)
- i specifies a unique transaction identifier (TId)

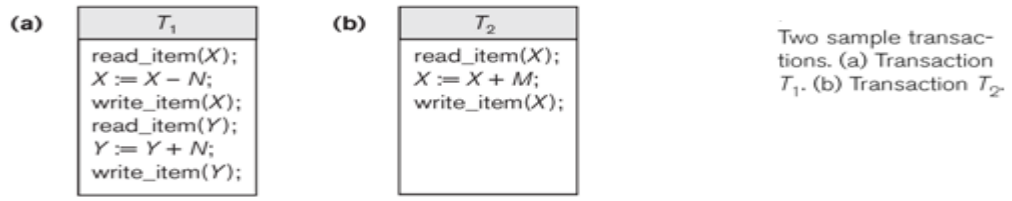


Figure 5.7: Two sample transactions, (a) Transaction T_1, (b) Transaction T_2

5.6.2 Why we need concurrency control?

Process of managing simultaneous operations on the database without having them interfere with one another.

- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Without Concurrency Control, problems may occur with concurrent transactions**:**

- **Lost Update Problem.**

Occurs when two transactions update the same data item, but both read the same original value before update as in Figure 5.8 (a).

- **The Temporary Update (or Dirty Read) Problem.**

This occurs when one transaction T1 updates a database item X, which is accessed (read) by another transaction T2; then T1 fails for some reason (Figure 5.8(b)); X was (read) by T2 before its value is changed back (rolled back or UNDONE) after T1 fails.

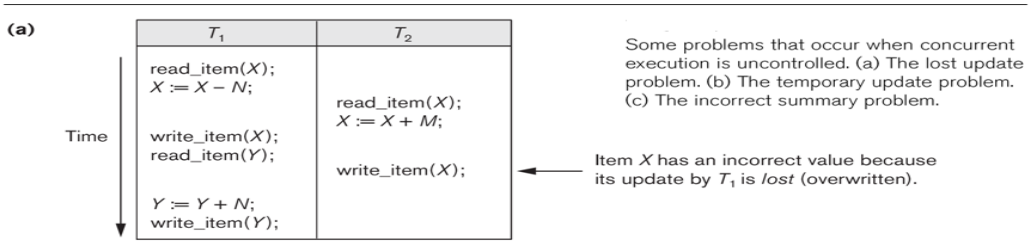


Figure 5.8 (a): Lost update problem

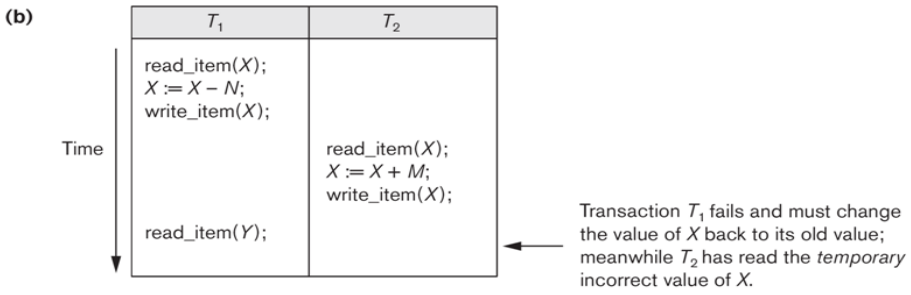


Figure 5.8 (b): Temporary update problem

- **The Incorrect Summary Problem .**

One transaction is calculating an aggregate summary function on a number of records (for example, sum (total) of all bank account balances) while other transactions are updating some of these records (for example, transferring a large amount between two accounts, see Figure 5.8(c)); the aggregate function may read some values before they are updated and others after they are updated.

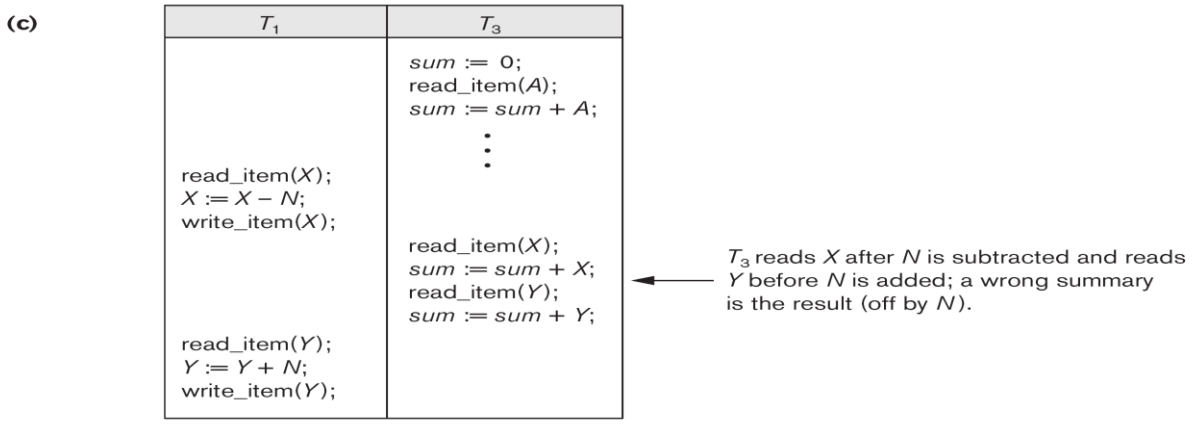


Figure 5.8 (c): Incorrect summary problem

- **The Unrepeatable Read Problem .**

A transaction T1 may read an item (say, available seats on a flight); later, T1 may read the same item again and get a different value because another transaction T2 has updated the item (reserved seats on the flight) between the two reads by T1

5.6.3 Why recovery is needed?

Causes of transaction failure:

1. A computer failure (system crash): A hardware or software error occurs during transaction execution. If the hardware crashes, the contents of the computer's internal main memory may be lost.
2. A transaction or system error : Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.
3. Local errors or exception conditions detected by the transaction:

- certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled - a programmed abort causes the transaction to fail.

4. Concurrency control enforcement: The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.
5. Disk failure: Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This kind of failure and item 6 are more severe than items 1 through 4.
6. Physical problems and catastrophes: This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

5.6.4 Transaction and System Concepts

A transaction is an atomic unit of work that is either completed in its entirety or not done at all. A transaction passes through several states (Figure 5.9, similar to process states in operating systems).

Transaction states:

- Active state (executing read, write operations)
- Partially committed state (ended but waiting for system checks to determine success or failure)
- Committed state (transaction succeeded)
- Failed state (transaction failed, must be rolled back)
- Terminated State (transaction leaves system)

State transition diagram illustrating the states for transaction execution.

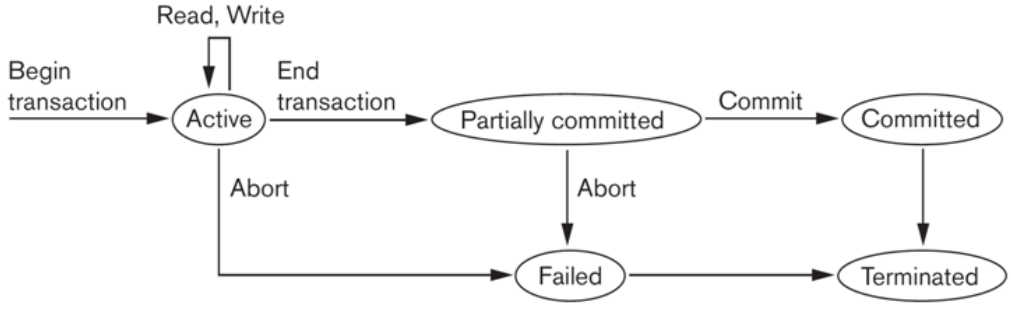


Figure 5.9: State transition diagram of transaction

DBMS Recovery Manager needs system to keep track of the following operations (in the system log file):

- begin_transaction: Start of transaction execution.
- read or write: Read or write operations on the database items that are executed as part of a transaction.
- end_transaction: Specifies end of read and write transaction operations have ended. System may still have to check whether the changes (writes) introduced by transaction can be *permanently applied to the database* (commit transaction); or whether the transaction has to be *rolled back* (abort transaction) because it violates concurrency control or for some other reason.
- Recovery manager keeps track of the following operations (cont.):
- **commit_transaction:** Signals *successful end* of transaction; any changes (writes) executed by transaction can be safely **committed** to the database and will not be undone.
- **abort_transaction (or rollback):** Signals transaction has *ended unsuccessfully*; any changes or effects that the transaction may have applied to the database must be *undone.*

System operations used during recovery :

- **undo(X):** Similar to rollback except that it applies to a single write operation rather than to a whole transaction.
- **redo(X):** This specifies that a *write operation* of a committed transaction must be *redone* to ensure that it has been applied permanently to the database on disk.

The System Log File

- Is an *append-only file* to keep track of all operations of all transactions *in the order in which they occurred*. This information is needed during recovery from failures
- Log is kept on disk - not affected except for disk or catastrophic failure
- As with other disk files, a *log main memory buffer* is kept for holding the records being appended until the whole buffer is appended to the end of the log file on disk
- Log is periodically backed up to archival storage (tape) to guard against catastrophic failures.

Types of records (entries) in log file:

- [start_transaction,T]: Records that transaction T has started execution.
- [write_item,T,X,old_value,new_value]: T has changed the value of item X from old_value to new_value.
- [read_item,T,X]: T has read the value of item X (not needed in many cases).
- [end_transaction,T]: T has ended execution
- [commit,T]: T has completed successfully, and committed.
- [abort,T]: T has been aborted.

The System Log :

- Protocols for recovery that avoid cascading rollbacks do not require that read operations be written to the system log; most recovery protocols fall in this category
- Strict protocols require simpler write entries that do not include new_value.

Commit Point of a Transaction:

- **Definition:** A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database has been recorded in the log file (on disk). The transaction is then said to be **committed**.

Characteristics of Commit Point of a Transaction:

- **Log file buffers:** Like database files on disk, whole disk blocks must be read or written to main memory buffers.
- For **log file**, the last disk block (or blocks) of the file will be in main memory buffers to easily append log entries at end of file.

- **Force writing the log buffer:** *before* a transaction reaches its commit point, any main memory buffers of the log that have not been written to disk yet must be copied to disk.
- Called **force-writing** the log buffers before committing a transaction.
- Needed to ensure that any write operations by the transaction are recorded in the log file *on disk* before the transaction commits.

5.7 ACID properties of Transaction:
Atomicity, Consistency, Isolation, Durability:

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: Even though transactions are executing concurrently, they should appear to be executed in isolation – that is, their final effect should be as if each transaction was executed in isolation from start to finish.
- **Durability or permanency**: Once a transaction is committed, its changes (writes) applied to the database must never be lost because of subsequent failure.

- **Atomicity**: Enforced by the recovery protocol.
- **Consistency preservation**: Specifies that each transaction does a correct action on the database *on its own*. Application programmers and DBMS constraint enforcement are responsible for this.
- **Isolation**: Responsibility of the concurrency control protocol.
- **Durability or permanency**: Enforced by the recovery protocol.

5.8 Schedules of Transactions

- **Transaction schedule (or history):** When transactions are executing concurrently in an interleaved fashion, the *order of execution* of operations from the various transactions forms what is known as a **transaction schedule** (or history).
- Figure 5.10, 5.11 show 4 possible schedules (A, B, C, D) of two transactions T1 and T2:
  - Order of operations from top to bottom
  - Each schedule includes *same operations*
  - Different *order of operations* in each schedule

Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

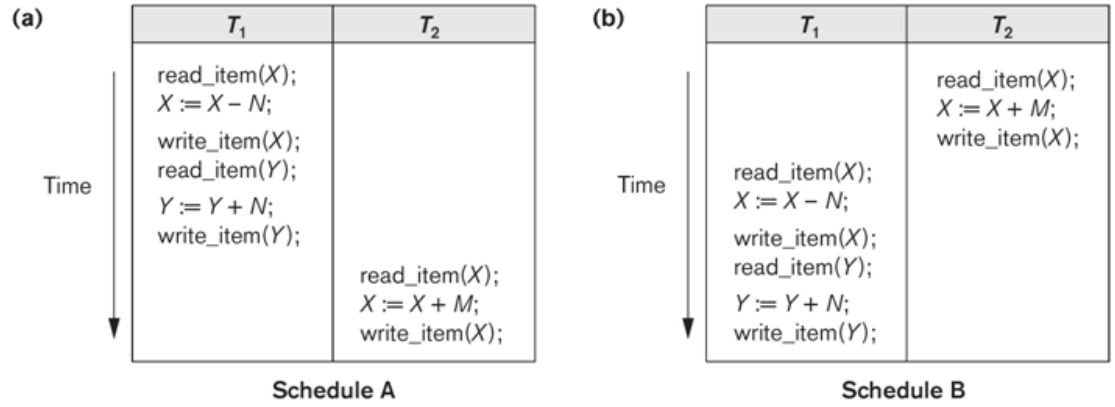


Figure 5.10: Examples of Serial schedules

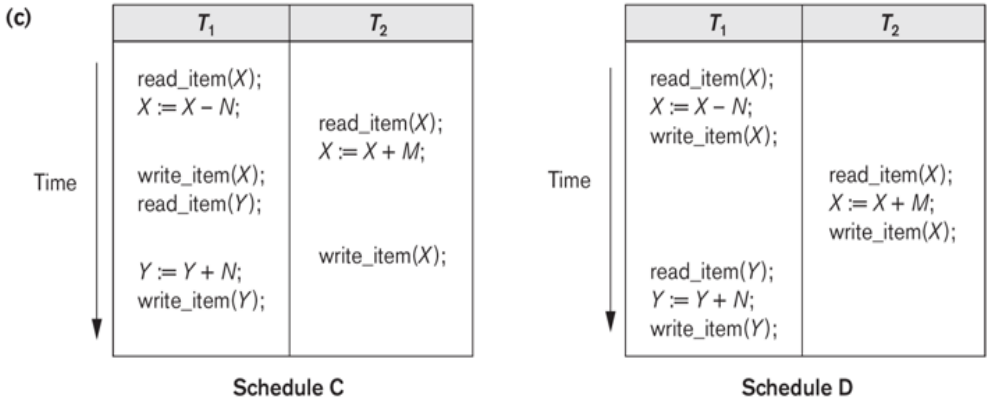


Figure 5.11: Examples of non-serial schedules

- Schedules can also be displayed in more compact notation
- Order of operations from left to right
- Include only read (r) and write (w) operations, with transaction id (1, 2, …) and item name (X, Y, …)
- Can also include other operations such as b (begin), e (end), c (commit), a (abort)
- Schedules in Figure 5.10, 5.11 would be displayed as follows:
  - Schedule A: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(x);
  - Schedule B: r2(X); w2(X); r1(X); w1(X); r1(Y); w1(Y);
  - Schedule C: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);
  - Schedule D: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y);

- Formal definition of a **schedule** (or **history**) S of n transactions T1, T2, ..., Tn :
  An ordering of all the operations of the transactions subject to the constraint that, for each transaction Ti that participates in S, the operations of Ti in S must appear *in the same order* in which they occur in Ti.
  Operations from other transactions Tj can be interleaved with the operations of Ti in S.

- For n transactions T1, T2, ..., Tn, where each Ti has mi read and write operations, the number of possible schedules is (! is *factorial* function):

(m1 + m2 + … + mn)! / ((m1)! * (m2)! * … * (mn)!)

- Generally very large number of possible schedules
- Some schedules are easy to recover from after a failure, while others are not
- Some schedules produce correct results, while others produce incorrect results
- Rest of chapter characterizes schedules by classifying them based on ease of recovery (recoverability) and correctness (serializability)

5.8.1 Characterizing Schedules based on Recoverability

Schedules classified into two main classes:

- **Recoverable schedule:** One where no *committed* transaction needs to be rolled back (aborted).

A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

- **Non-recoverable schedule:** A schedule where a committed transaction may have to be rolled back during recovery.

This violates **Durability** from ACID properties (a committed transaction cannot be rolled back) and so non-recoverable schedules *should not be allowed*.

- **Example:** Schedule A below is **non-recoverable** because T2 reads the value of X that was written by T1, but then T2 commits before T1 commits or aborts
- To make it **recoverable**, the commit of T2 (c2) must be delayed until T1 either commits, or aborts (Schedule B)
- If T1 commits, T2 can commit
- If T1 aborts, T2 must also abort because it read a value that was written by T1; this value must be undone (reset to its old value) when T1 is aborted
  - known as *cascading rollback*
- Schedule A: r1(X); w1(X); r2(X); w2(X); c2; r1(Y); w1(Y); c1 (or a1)
- Schedule B: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y); c1 (or a1); ...

Recoverable schedules can be further refined**:**

- **Cascadeless schedule:** A schedule in which a transaction T2 cannot read an item X until the transaction T1 that last wrote X has committed.
- The set of cascadeless schedules is a *subset of* the set of recoverable schedules.

Schedules requiring cascaded rollback: A schedule in which an uncommitted transaction T2 that read an item that was written by a failed transaction T1 must be rolled back.

- **Example:** Schedule B below is **not cascadeless** because T2 reads the value of X that was written by T1 before T1 commits
- If T1 aborts (fails), T2 must also be aborted (rolled back) resulting in *cascading rollback*
- To make it **cascadeless**, the r2(X) of T2 must be delayed until T1 commits (or aborts and rolls back the value of X to its previous value) – see Schedule C
- Schedule B: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y); c1 (or a1);
- Schedule C: r1(X); w1(X); r1(Y); w1(Y); c1; r2(X); w2(X); ...

5.8.2 Cascadeless schedules can be further refined**:**

- **Strict schedule:** A schedule in which a transaction T2 can neither read *nor write* an item X until the transaction T1 that last wrote X has committed.
- The set of strict schedules is a *subset of* the set of cascadeless schedules.
- If *blind writes* are not allowed, all cascadeless schedules are also strict
  Blind write: A write operation w2(X) that is not preceded by a read r2(X).
-
- **Example:** Schedule C below is **cascadeless** and also **strict** (because it has no blind writes)
- Schedule D is cascadeless, but not strict (because of the blind write w3(X), which writes the value of X before T1 commits)
- To make it strict, w3(X) must be delayed until after T1 commits – see Schedule E
- Schedule C: r1(X); w1(X); r1(Y); w1(Y); c1; r2(X); w2(X); …
- Schedule D: r1(X); w1(X); w3(X); r1(Y); w1(Y); c1; r2(X); w2(X); …
- Schedule E: r1(X); w1(X); r1(Y); w1(Y); c1; w3(X); r2(X); w2(X); …

5.8.3 Characterizing Schedules based on Serializability

- Among the large set of possible schedules, we want to characterize which schedules are *guaranteed to give a correct result*
- The consistency preservation property of the ACID properties states that: each transaction if executed on its own (from start to finish) will transform a consistent state of the database into another consistent state
- Hence, each transaction is *correct* on its own

- Serial schedule: A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively (without interleaving of operations from other transactions) in the schedule. Otherwise, the schedule is called nonserial.
- Based on the consistency preservation property, *any serial schedule will produce a correct result* (assuming no inter-dependencies among different transactions)

- Serial schedules are *not feasible* for performance reasons:
  - No interleaving of operations
  - Long transactions force other transactions to wait
  - System cannot switch to other transaction when a transaction is waiting for disk I/O or any other event
  - Need to allow concurrency with interleaving without sacrificing correctness

- Serializable schedule: A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.
- There are (n)! serial schedules for n transactions – a serializable schedule can be equivalent to *any of the serial schedules*

In serializability, ordering of read/writes is important:
(a) If two transactions only read a data item, they do not conflict and order is not important.
(b) If two transactions either read or write completely separate data items, they do not conflict and order is not important.
(c) If one transaction writes a data item and another reads or writes same data item, order of execution is important.

5.8.4 Equivalence of Schedules

- Result equivalent: Two schedules are called result equivalent if they produce the same final state of the database.
- Difficult to determine without *analyzing the internal operations of the transactions*, which is not feasible in general.
- May also get result equivalence *by chance* for a particular input parameter even though schedules *are not equivalent in general* (see Figure 5.12)

Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

S_1	S_2

read_item(X);
$X := X + 10$;
write_item(X);
	read_item(X);
	$X := X * 1.1$;
	write_item (X);

Figure 5.12: Examples of equivalent schedules

- **Conflict equivalent**: Two schedules are conflict equivalent if the relative order of *any two conflicting operations* is the same in both schedules.
- Commonly used definition of schedule equivalence
- Two operations are **conflicting** if:
  - They access the same data item X
  - They are from two different transactions
  - At least one is a write operation

- Read-Write conflict example: r1(X) and w2(X)
- Write-write conflict example: w1(Y) and w2(Y)
- Changing the order of conflicting operations generally *causes a different outcome*
- **Example:** changing r1(X); w2(X) to w2(X); r1(X) means that T1 will read *a different value for X*
- **Example:** changing w1(Y); w2(Y) to w2(Y); w1(Y) means that the final value for Y in the database can be different
- Note that read operations are **not conflicting**; changing r1(Z); r2(Z) to r2(Z); r1(Z) does not change the outcome.

5.8.5 Characterizing Schedules Based on Serializability

Conflict equivalence of schedules is used to determine which schedules are correct in general (serializable)
A schedule S is said to be **serializable** if it is conflict equivalent to some serial schedule S'.

- A serializable schedule is considered to be correct because it is equivalent to a serial schedule, and any serial schedule is considered to be correct
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution and interleaving of operations from different transactions.

- Serializability is generally hard to check at run-time:
  - Interleaving of operations is generally handled by the operating system through the process scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved
  - Transactions are continuously started and terminated

Testing for conflict serializability
Algorithm :
- Looks at only r(X) and w(X) operations in a schedule
- Constructs a precedence graph (serialization graph) – **one node for each transaction**, plus directed edges
- An **edge is created** from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
- The schedule is serializable if and only if the precedence graph **has no cycles**.

Algorithm . Testing Conflict Serializability of a Schedule S

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

Figure 5.13: Testing for conflict serializability

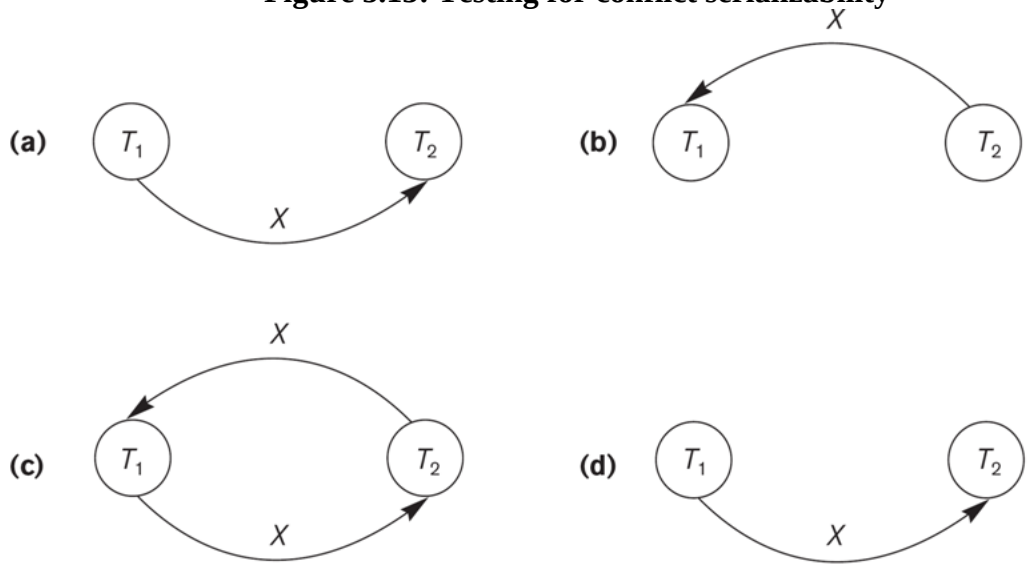


Figure
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Figure 5.14: Testing for conflict serializability using graph

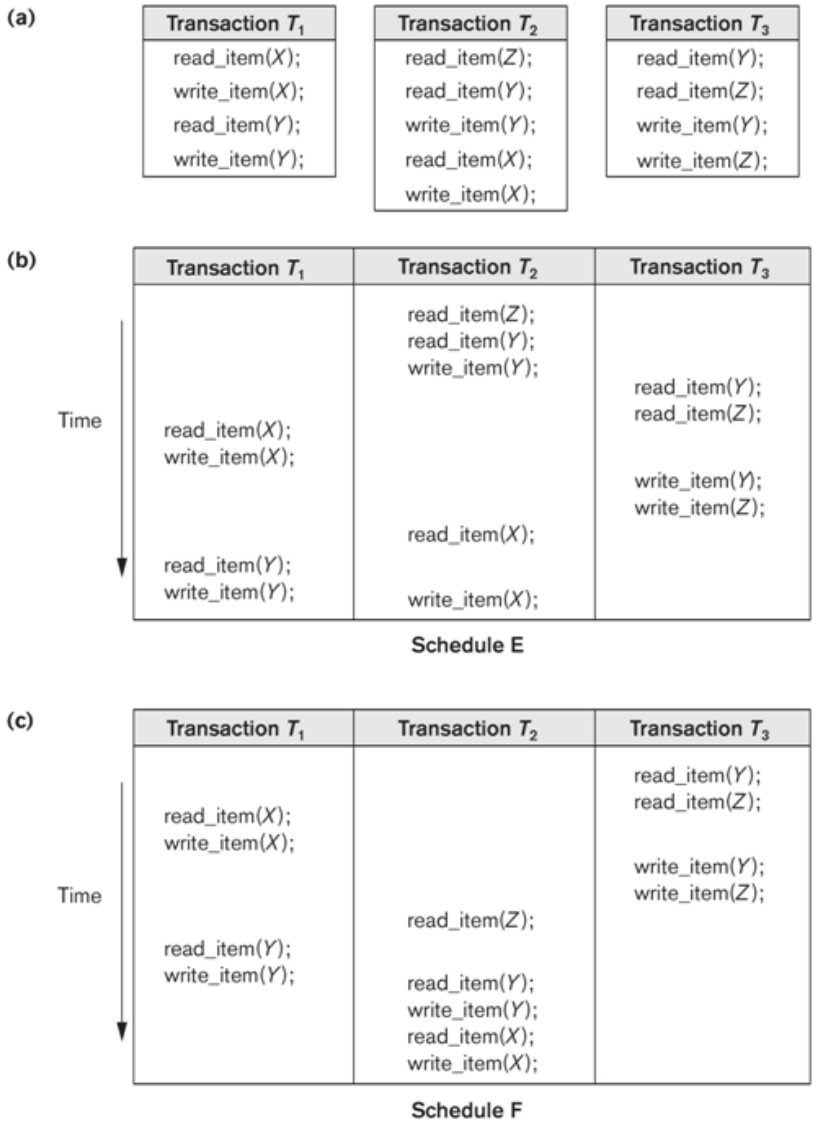


Figure
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule F.

Figure 5.15: Testing for conflict serializability using example

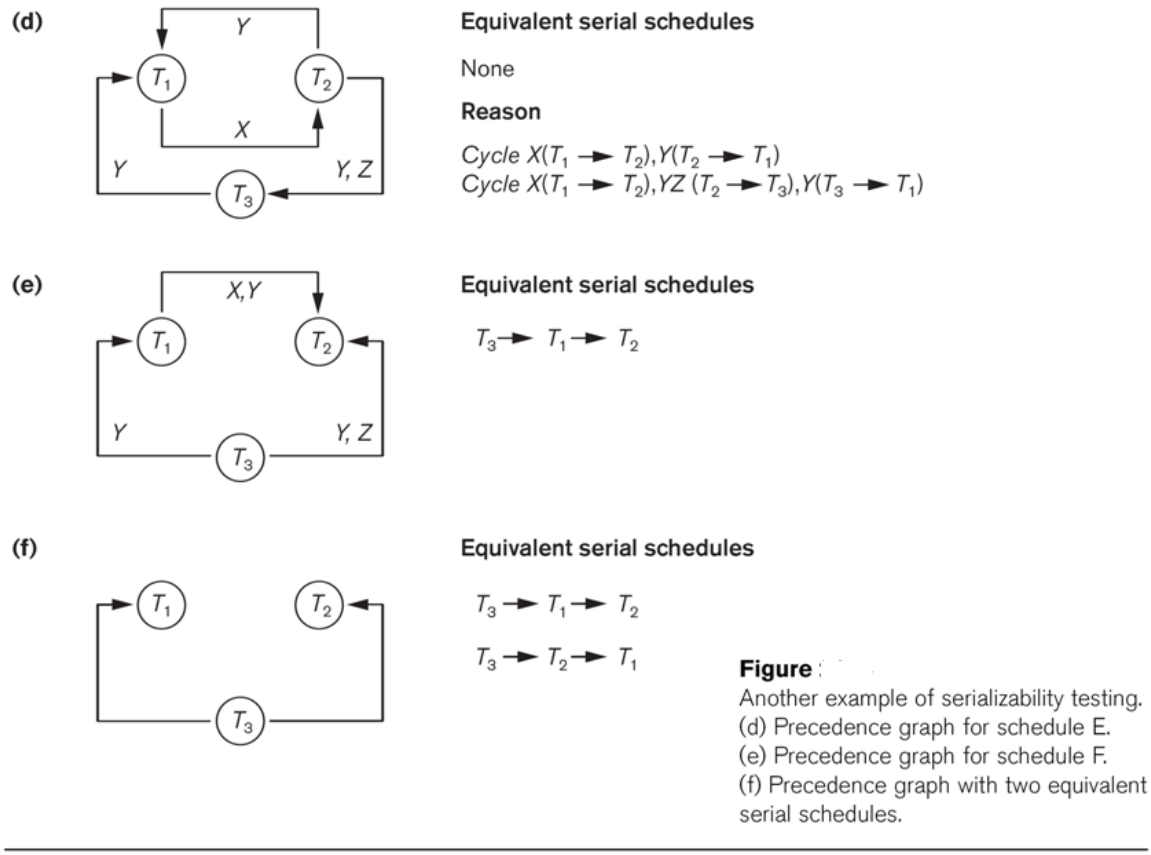


Figure 5.16: Testing for conflict serializability using another example

- **View equivalence**: A less restrictive definition of equivalence of schedules than conflict serializability *when blind writes are allowed*
- **View serializability:** definition of serializability based on view equivalence. A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

Two schedules S_1 and S_2 are view equivalent if:

- For each data item x, if T_i reads initial value of x in S_1, T_i must also read initial value of x in S_2.
- For each read on x by T_i in S_1, if value read by x is written by T_j, T_i must also read value of x produced by T_j in S_2.
- For each data item x, if last write on x performed by T_i in S_1, same transaction must perform final write on x in S_2.

Two schedules are said to be view equivalent if the following three conditions hold:

- The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
- For any operation Ri(X) of Ti in S, if the value of X read was written by an operation Wj(X) of Tj (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation Ri(X) of Ti in S'.
- If the operation Wk(Y) of Tk is the last operation to write item Y in S, then Wk(Y) of Tk must also be the last operation to write item Y in S'.

The premise behind view equivalence:

- Each read operation of a transaction reads the result of *the same write operation* in both schedules.
- "The view": the read operations are said to see the *the same view* in both schedules.

- The final write operation on each item is the same on both schedules resulting in the same final database state in case of blind writes

Relationship between view and conflict equivalence:

- The two are same under constrained write assumption (no blind writes allowed)
- Conflict serializability is stricter than view serializability when blind writes occur (a schedule that is view serializable is not necessarily conflict serialiable.
- Any conflict serializable schedule is also view serializable, but not vice versa.

Consider the following schedule of three transactions
T1: r1(X); w1(X); T2: w2(X); and T3: w3(X):
Schedule Sa: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;
In Sa, the operations w2(X) and w3(X) are blind writes, since T2 and T3 do not read the value of X.
Sa is view serializable, since it is view equivalent to the serial schedule T1, T2, T3. However, Sa is not conflict serializable, since it is not conflict equivalent to any serial schedule.

Other Types of Equivalence of Schedules

- Under special semantic constraints, schedules that are otherwise not conflict serializable may work correctly
- Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly; known as debit-credit transactions

Example: bank credit/debit transactions on a given item are separable and commutative.
Consider the following schedule S for the two transactions:
Sh : r1(X); w1(X); r2(Y); w2(Y); r1(Y); w1(Y); r2(X); w2(X);
Using conflict serializability, it is not serializable.
However, if it came from a (read,update, write) sequence as follows:
r1(X); X := X – 10; w1(X); r2(Y); Y := Y – 20; w2(Y); r1(Y);
Y := Y + 10; w1(Y); r2(X); X := X + 20; w2(X);
Sequence explanation: debit, debit, credit, credit.
It is a correct schedule for the given semantics

View serializability offers less stringent definition of schedule equivalence than conflict serializability.
Two schedules S_1 and S_2 are view equivalent if:

- For each data item x, if T_i reads initial value of x in S_1, T_i must also read initial value of x in S_2.
- For each read on x by T_i in S_1, if value read by x is written by T_j, T_i must also read value of x produced by T_j in S_2.
- For each data item x, if last write on x performed by T_i in S_1, same transaction must perform final write on x in S_2.

Schedule is view serializable if it is view equivalent to a serial schedule.
Every conflict serializable schedule is view serializable, although converse is not true.
It can be shown that any view serializable schedule that is not conflict serializable contains one or more blind writes.
In general, testing whether schedule is serializable is NP-complete.

Time	T_{11}	T_{12}	T_{13}
t_1	begin_transaction		
t_2	read(bal_x)		
t_3		begin_transaction	
t_4		write(bal_x)	
t_5		commit	
t_6	write(bal_x)		
t_7	commit		
t_8			begin_transaction
t_9			write(bal_x)
t_{10}			commit

Figure 5.17: Example of view serializability

5.9 Recoverability

Serializability identifies schedules that maintain database consistency, assuming no transaction fails.
It could also examine recoverability of transactions within schedule.
If transaction fails, atomicity requires effects of transaction to be undone.
Durability states that once transaction commits, its changes cannot be undone (without running another, compensating, transaction).

Recoverable Schedule

A schedule where, for each pair of transactions T_i and T_j, if T_j reads a data item previously written by T_i, then the commit operation of T_i precedes the commit operation of T_j.

5.10 Concurrency Control Techniques

Two basic concurrency control techniques:

- Locking,
- Timestamping.

Both are conservative approaches: delay transactions in case they conflict with other transactions.
Optimistic methods assume conflict is rare and only check for conflicts at commit.

5.10.1 Locking

- A lock is a mechanism to control concurrent access to a data item
- Transaction uses locks to deny access to other transactions and so prevent incorrect updates.
- Most widely used approach to ensure serializability.
- Generally, a transaction must claim a *shared* (*read*) or *exclusive* (*write*) lock on a data item before read or write.
- Lock prevents another transaction from modifying item or even reading it, in the case of a write lock.

Locking - Basic Rules

Data items can be locked in two modes :

1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using lock-X instruction.
2. *shared (S) mode*. Data item can only be read. S-lock is requested using lock-S instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

- If transaction has shared lock on item, can read but not update item.
- If transaction has exclusive lock on item, can both read and update item.
- Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
- Exclusive lock gives transaction exclusive access to that item.
- Some systems allow transaction to upgrade read lock to an exclusive lock, or downgrade exclusive lock to a shared lock.

For two transactions above, a valid schedule using these rules is:
S = {write_lock(T_9, bal_x), read(T_9, bal_x), write(T_9, bal_x), unlock(T_9, bal_x), write_lock(T_{10}, bal_x), read(T_{10}, bal_x), write(T_{10}, bal_x), unlock(T_{10}, bal_x), write_lock(T_{10}, bal_y), read(T_{10}, bal_y), write(T_{10}, bal_y), unlock(T_{10}, bal_y), commit(T_{10}), write_lock(T_9, bal_y), read(T_9, bal_y), write(T_9, bal_y), unlock(T_9, bal_y), commit(T_9) }

If at start, bal_x = 100, bal_y = 400, result should be:
- bal_x = 220, bal_y = 330, if T_9 executes before T_{10}, or
- bal_x = 210, bal_y = 340, if T_{10} executes before T_9.

However, result gives bal_x = 220 and bal_y = 340.
S is not a serializable schedule.

Problem is that transactions release locks too soon, resulting in loss of total isolation and atomicity.
To guarantee serializability, need an additional protocol concerning the positioning of lock and unlock operations in every transaction.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

Figure 5.18: Example of Lock

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Example of a transaction performing locking:

T_2: **lock-S***(A)*;

read *(A)*;
unlock*(A)*;
lock-S*(B)*;
read *(B)*;
unlock*(B)*;
display*(A+B)*

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

Consider the partial schedule

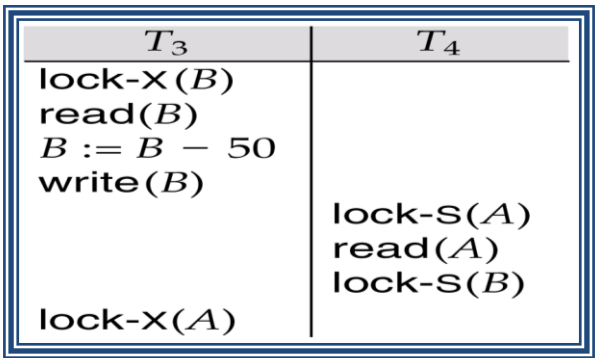


Figure 5.19: Example of partial schedule

- Neither T_3 nor T_4 can make progress — executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.
- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

5.10.2 Two-Phase Locking (2PL)

Transaction follows 2PL protocol if all locking operations precede first unlock operation in the transaction.

Two phases for transaction:

- Growing phase - acquires all locks but cannot release any locks.
- Shrinking phase - releases locks but cannot acquire any new locks.

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.
- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
- Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

Two-phase locking with lock conversions:

First Phase:
- can acquire a lock-S on item
- can acquire a lock-X on item
- can convert a lock-S to a lock-X (upgrade)

Second Phase:
- can release a lock-S
- can release a lock-X
- can convert a lock-X to a lock-S (downgrade)

This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(*D*) is processed as:

```
if Ti has a lock on D
  then
       read(D)
  else begin
         if necessary wait until no other
            transaction has a lock-X on D
         grant Ti a  lock-S on D;
         read(D)
       end
```

- **write***(D)* is processed as:

```
if Ti has a  lock-X on D
  then
    write(D)
  else begin
     if necessary wait until no other trans. has any lock on D,
     if Ti has a lock-S on D
        then
          upgrade lock on D  to lock-X
        else
          grant Ti a lock-X on D
        write(D)
   end;
```

- All locks are released after commit or abort

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

5.10.3 Lock Table

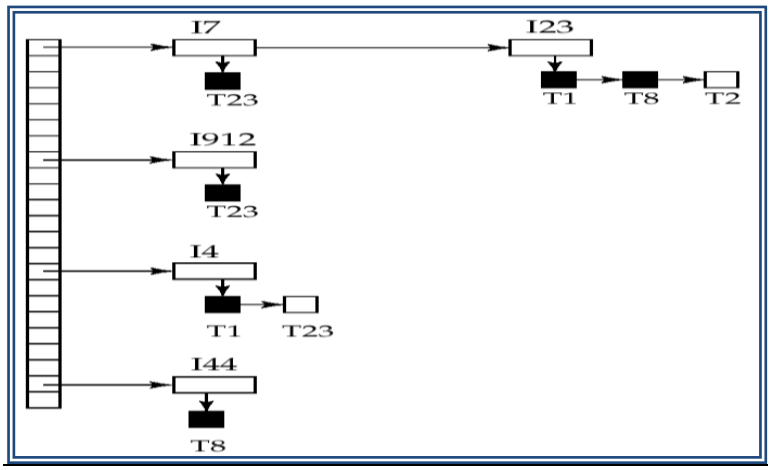


Figure 5.20: Example of Lock table

- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
    - lock manager may keep a list of locks held by each transaction, to implement this efficiently.

5.11 Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, ..., d_h\}$ of all data items.
    - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
    - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol

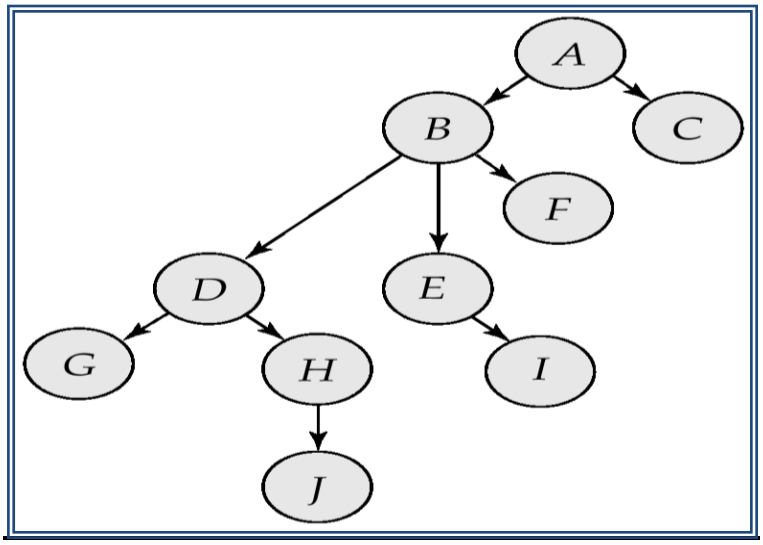


Figure 5.21: Example of Tree protocol

1. Only exclusive locks are allowed.
2. The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
    - shorter waiting times, and increase in concurrency
    - protocol is deadlock-free, no rollbacks are required
- Drawbacks
    - Protocol does not guarantee recoverability or cascade freedom
        - Need to introduce commit dependencies to ensure recoverability
    - Transactions may have to lock data items that they do not access.
        - increased locking overhead, and additional waiting time
        - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

5.12 Cascading Rollback

If *every* transaction in a schedule follows 2PL, schedule is serializable.
However, problems can occur with interpretation of when locks can be released.

Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock(bal_x)		
t_3	read(bal_x)		
t_4	read_lock(bal_y)		
t_5	read(bal_y)		
t_6	$bal_x = bal_y + bal_x$		
t_7	write(bal_x)		
t_8	unlock(bal_x)	begin_transaction	
t_9	⋮	write_lock(bal_x)	
t_{10}	⋮	read(bal_x)	
t_{11}	⋮	$bal_x = bal_x + 100$	
t_{12}	⋮	write(bal_x)	
t_{13}	⋮	unlock(bal_x)	
t_{14}	⋮	⋮	
t_{15}	rollback	⋮	
t_{16}		⋮	begin_transaction
t_{17}		⋮	read_lock(bal_x)
t_{18}		rollback	⋮
t_{19}			rollback

Figure5.22: Cascading rollback

- Transactions conform to 2PL.
- T_{14} aborts.
- Since T_{15} is dependent on T_{14}, T_{15} must also be rolled back. Since T_{16} is dependent on T_{15}, it too must be rolled back.
- This is called *cascading rollback*.
- To prevent this with 2PL, leave release of *all* locks until end of transaction.

Concurrency Control with Index Structures

Could treat each page of index as a data item and apply 2PL.
However, as indexes will be frequently accessed, particularly higher levels, this may lead to high lock contention.
Can make two observations about index traversal:

- Search path starts from root and moves down to leaf nodes but search never moves back up tree. Thus, once a lower-level node has been accessed, higher-level nodes in that path will not be used again.

- When new index value (key and pointer) is being inserted into a leaf node, then if node is not full, insertion will not cause changes to higher-level nodes.

Suggests only have to exclusively lock leaf node in such a case, and only exclusively lock higher-level nodes if node is full and has to be split.

Thus, can derive following locking strategy:

- For searches, obtain shared locks on nodes starting at root and proceeding downwards along required path. Release lock on node once lock has been obtained on the child node.
- For insertions, conservative approach would be to obtain exclusive locks on all nodes as we descend tree to the leaf node to be modified.
- For more optimistic approach, obtain shared locks on all nodes as we descend to leaf node to be modified, where obtain exclusive lock. If leaf node has to split, upgrade shared lock on parent to exclusive lock. If this node also has to split, continue to upgrade locks at next higher level.

5.13.Deadlock

An impasse that may result when two (or more) transactions are each waiting for locks held by the other to be released.

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	write_lock(bal_x)	begin_transaction
t_3	read(bal_x)	write_lock(bal_y)
t_4	$bal_x = bal_x - 10$	read(bal_y)
t_5	write(bal_x)	$bal_y = bal_y + 100$
t_6	write_lock(bal_y)	write(bal_y)
t_7	WAIT	write_lock(bal_x)
t_8	WAIT	WAIT
t_9	WAIT	WAIT
t_{10}	⋮	WAIT
t_{11}	⋮	⋮

Figure5.23: Deadlock

Only one way to break deadlock: abort one or more of the transactions.
Deadlock should be transparent to user, so DBMS should restart transaction(s).
Three general techniques for handling deadlock:

- Timeouts.
- Deadlock prevention.
- Deadlock detection and recovery.

Timeouts

Transaction that requests lock will only wait for a system-defined period of time.
If lock has not been granted within this period, lock request times out.
In this case, DBMS assumes transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.

5.13.1 Deadlock Prevention

DBMS looks ahead to see if transaction would cause deadlock and never allows deadlock to occur.
Could order transactions using transaction timestamps:

- **Wait-Die** - only an older transaction can wait for younger one, otherwise transaction is aborted (*dies*) and restarted with same timestamp.
- **Wound-Wait** - only a younger transaction can wait for an older one. If older transaction requests lock held by younger one, younger one is aborted (wounded).

5.13.2 Deadlock Detection and Recovery

DBMS allows deadlock to occur but recognizes it and breaks it.
Usually handled by construction of wait-for graph (WFG) showing transaction dependencies:

- Create a node for each transaction.
- Create edge T_i -> T_j, if T_i waiting to lock item locked by T_j.

Deadlock exists if and only if WFG contains cycle.

WFG is created at regular intervals.

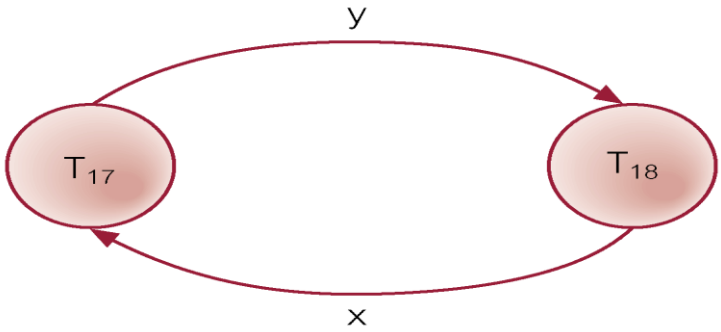


Figure 5.24: Wait-For-Graph (WFG)

Recovery from Deadlock Detection

Several issues:
- choice of deadlock victim;
- how far to roll a transaction back;
- avoiding starvation.

5.13.3 Timestamping

Transactions ordered globally so that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.
Conflict is resolved by rolling back and restarting transaction.
No locks so no deadlock.

Timestamp

- A unique identifier created by DBMS that indicates relative starting time of a transaction.
- Can be generated by using system clock at time transaction started, or by incrementing a logical counter every time a new transaction starts.

- Read/write proceeds only if *last update on that data item* was carried out by an older transaction.
- Otherwise, transaction requesting read/write is restarted and given a new timestamp.
- Also timestamps for data items:
  - read-timestamp - timestamp of last transaction to read item;
  - write-timestamp - timestamp of last transaction to write item.

Timestamping - Read(x)

Consider a transaction T with timestamp ts(T):

ts(T) < write_timestamp(x)
- x already updated by younger (later) transaction.
- Transaction must be aborted and restarted with a new timestamp.

ts(T) < read_timestamp(x)
- x already read by younger transaction.
- Roll back transaction and restart it using a later timestamp.

Timestamping - Write(x)

ts(T) < write_timestamp(x)

- x already written by younger transaction.
- Write can safely be ignored - *ignore obsolete write* rule.
- Otherwise, operation is accepted and executed.

Example – Basic Timestamp Ordering

Time	Op	T_{19}	T_{20}	T_{21}
t_1		begin_transaction		
t_2	read(bal_x)	read(bal_x)		
t_3	$bal_x = bal_x + 10$	$bal_x = bal_x + 10$		
t_4	write(bal_x)	write(bal_x)	begin_transaction	
t_5	read(bal_y)		read(bal_y)	
t_6	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	begin_transaction
t_7	read(bal_y)			read(bal_y)
t_8	write(bal_y)		write(bal_y)+	
t_9	$bal_y = bal_y + 30$			$bal_y = bal_y + 30$
t_{10}	write(bal_y)			write(bal_y)
t_{11}	$bal_z = 100$			$bal_z = 100$
t_{12}	write(bal_z)			write(bal_z)
t_{13}	$bal_z = 50$	$bal_z = 50$		commit
t_{14}	write(bal_z)	write(bal_z)‡	begin_transaction	
t_{15}	read(bal_y)	commit	read(bal_y)	
t_{16}	$bal_y = bal_y + 20$		$bal_y = bal_y + 20$	
t_{17}	write(bal_y)		write(bal_y)	
t_{18}			commit	

+ At time t_8, the write by transaction T_{20} violates the first timestamping write rule described above and therefore is aborted and restarted at time t_{14}.
‡ At time t_{14}, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21} at time t_{12}.

Figure 5.25: Example – Basic Timestamp Ordering

5.13.4 Multiversion Timestamp Ordering

- Versioning of data can be used to increase concurrency.
- Basic timestamp ordering protocol assumes only one version of data item exists, and so only one transaction can access data item at a time.
- Can allow multiple transactions to read and write different versions of same data item, and ensure each transaction sees consistent set of versions for all data items it accesses.
- In multiversion concurrency control, each write operation creates new version of data item while retaining old version.
- When transaction attempts to read data item, system selects one version that ensures serializability.
- Versions can be deleted once they are no longer required.

5.13.5 Optimistic Techniques

- Based on assumption that conflict is rare and more efficient to let transactions proceed without delays to ensure serializability.
- At commit, check is made to determine whether conflict has occurred.
- If there is a conflict, transaction must be rolled back and restarted.
- Potentially allows greater concurrency than traditional protocols.

- Three phases:
  - Read
  - Validation
  - Write

Optimistic Techniques - Read Phase

- Extends from start until immediately before commit.
- Transaction reads values from database and stores them in local variables. Updates are applied to a local copy of the data.

Optimistic Techniques - Validation Phase

- Follows the read phase.
- For read-only transaction, checks that data read are still current values. If no interference, transaction is committed, else aborted and restarted.
- For update transaction, checks transaction leaves database in a consistent state, with serializability maintained.

Optimistic Techniques - Write Phase

- Follows successful validation phase for update transactions.
- Updates made to local copy are applied to the database.

5.13.6 Granularity of Data Items

- Size of data items chosen as unit of protection by concurrency control protocol.
- Ranging from coarse to fine:
  - The entire database.
  - A file.
  - A page (or area or database spaced).
  - A record.
  - A field value of a record.
- Tradeoff:
  - coarser, the lower the degree of concurrency;
  - finer, more locking information that is needed to be stored.
- Best item size depends on the types of transactions.

Hierarchy of Granularity

- Could represent granularity of locks in a hierarchical structure.
- Root node represents entire database, level 1s represent files, etc.
- When node is locked, all its descendants are also locked.
- DBMS should check hierarchical path before granting lock.
- Intention lock could be used to lock all ancestors of a locked node.
- Intention locks can be read or write. Applied top-down, released bottom-up.

Lock compatibility table for multiple-granularity locking.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

✓ = compatible, ✗ = incompatible

Figure 5.26: Lock compatibility for multiple-granularity locking

Levels of Locking

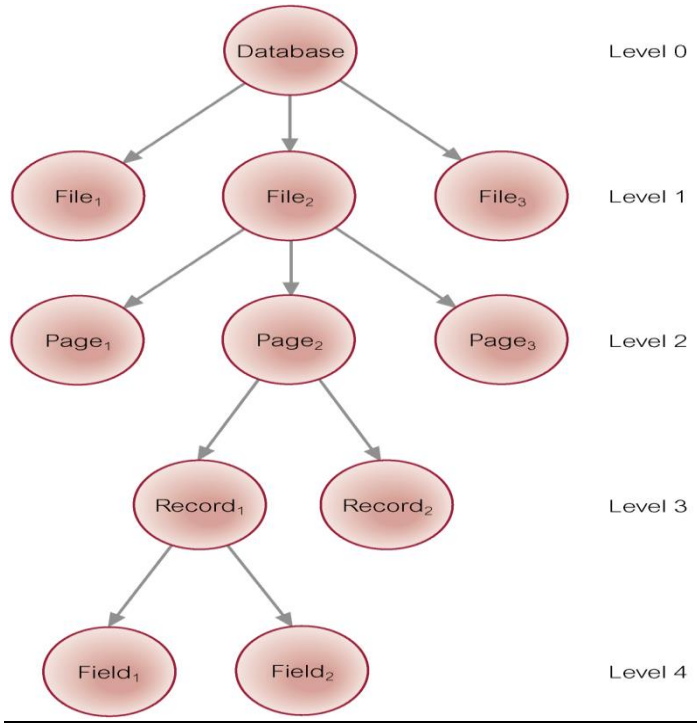


Figure 5.27: Levels of Locking

5.14 Database Recovery

Process of restoring database to a correct state in the event of a failure.
Need for Recovery Control

- Two types of storage: volatile (main memory) and nonvolatile.
- Volatile storage does not survive system crashes.
- Stable storage represents information that has been replicated in several nonvolatile storage media with independent failure modes.

Types of Failures

- System crashes, resulting in loss of main memory.
- Media failures, resulting in loss of parts of secondary storage.
- Application software errors.
- Natural physical disasters.

- Carelessness or unintentional destruction of data or facilities.
- Sabotage.

5.14.1 Transactions and Recovery

- Transactions represent basic unit of recovery.
- Recovery manager responsible for atomicity and durability.
- If failure occurs between commit and database buffers being flushed to secondary storage then, to ensure durability, recovery manager has to *redo* (*rollforward*) transaction's updates.
- DBMS starts at time t_0, but fails at time t_f. Assume data for transactions T_2 and T_3 have been written to secondary storage.
- T_1 and T_6 have to be undone. In absence of any other information, recovery manager has to redo T_2, T_3, T_4, and T_5.

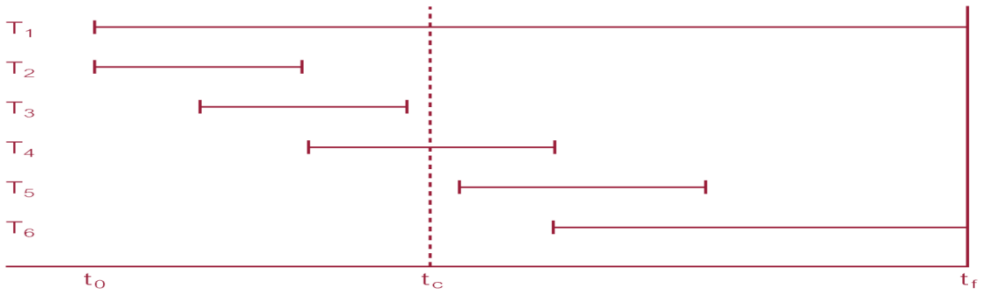


Figure 5.28: Example of Recovery

Recovery Facilities

DBMS should provide following facilities to assist with recovery:

- Backup mechanism, which makes periodic backup copies of database.
- Logging facilities, which keep track of current state of transactions and database changes.
- Checkpoint facility, which enables updates to database in progress to be made permanent.
- Recovery manager, which allows DBMS to restore database to consistent state following a failure.

5.14.2 Log File

- Contains information about all updates to database:
  - Transaction records.
  - Checkpoint records.
- Often used for other purposes (for example, auditing).
- Transaction records contain:
  - Transaction identifier.
  - Type of log record, (transaction start, insert, update, delete, abort, commit).
  - Identifier of data item affected by database action (insert, delete, and update operations).
  - Before-image of data item.

– After-image of data item.
– Log management information.

Tid	Time	Operation	Object	Before image	After image	pPtr	nPtr
T1	10:12	START				0	2
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
T1	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	0
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

Figure 5.29: Sample Log file

Log File

- Log file may be duplexed or triplexed.
- Log file sometimes split into two separate random-access files.
- Potential bottleneck; critical in determining overall performance.

5.14.3 Checkpointing

Checkpoint

Point of synchronization between database and log file. All buffers are force-written to secondary storage.

- Checkpoint record is created containing identifiers of all active transactions.
- When failure occurs, redo all transactions that committed since the checkpoint and undo all transactions active at time of crash.
- In previous example of figure 5.25, with checkpoint at time tc, changes made by T2 and T3 have been written to secondary storage.
- Thus:
  only redo T4 and T5,
  undo transactions T1 and T6.

5.14.4 Recovery Techniques

- If database has been damaged:

  Need to restore last backup copy of database and reapply updates of committed transactions using log file.
- If database is only inconsistent:

  Need to undo changes that caused inconsistency. May also need to redo some transactions to ensure updates reach secondary storage.
  Do not need backup, but can restore database using before- and after-images in the log file.

Main Recovery Techniques

Three main recovery techniques:

- Deferred Update
- Immediate Update
- Shadow Paging

5.14.5 Deferred Update

- Updates are not written to the database until after a transaction has reached its commit point.
- If transaction fails before commit, it will not have modified database and so no undoing of changes required.
- May be necessary to redo updates of committed transactions as their effect may not have reached database.

5.14.6 Immediate Update

- Updates are applied to database as they occur.
- Need to redo updates of committed transactions following a failure.
- May need to undo effects of transactions that had not committed at time of failure.
- Essential that log records are written before write to database. *Write-ahead log protocol*.
- If no "transaction commit" record in log, then that transaction was active at failure and must be undone.
- Undo operations are performed in reverse order in which they were written to log.

5.14.7 Shadow Paging

- Maintain two page tables during life of a transaction: *current* page and *shadow* page table.
- When transaction starts, two pages are the same.
- Shadow page table is never changed thereafter and is used to restore database in event of failure.
- During transaction, current page table records all updates to database.
- When transaction completes, current page table becomes shadow page table.

MODULE 5 MCQ and Short type Problem

Multiple choice questions (MCQ)

1. A transaction is delimited by statements (or function calls) of the form _____
a) Begin transaction and end transaction
b) Start transaction and stop transaction
c) Get transaction and post transaction
d) Read transaction and write transaction
Ans. (a)
2. Identify the characteristics of transactions
a) Atomicity
b) Durability

c) Isolation
d) All of the mentioned
Ans. (d)
3. Which of the following has “all-or-none” property ?
a) Atomicity
b) Durability
c) Isolation
d) All of the mentioned
View Answer
Answer: (a)
4. A _____ consists of a sequence of query and/or update statements.
a) Transaction
b) Commit
c) Rollback
d) Flashback
View Answer
Answer: a
5. In order to undo the work of transaction after last commit which one should be used ?
a) View
b) Commit
c) Rollback
d) Flashback
View Answer
Answer: c

Short question

1. What is 2PL?
2. Explain ACID properties in transaction.
3. Explain different transaction states.

Long type question

1. What is deadlock? What are the conditions for occurring deadlock in a system? Explain deadlock avoidance methods.
2. What is schedule in dbms? Explain view serializability and conflict serializability in dbms.
3. Explain timestamp based protocol. What are the different recovery techniques?

Text Books:

1. Henry F. Korth and Silberschatz Abraham, “Database System Concepts”, Mc.Graw Hill.
2. Elmasri Ramez and Novathe Shamkant, “Fundamentals of Database Systems”, Benjamin Cummings Publishing. Company.
3. Date C. J., “Introduction to Database Management”, Vol. I, II, III, Addison Wesley.
4. Jain: Advanced Database Management System CyberTech

Module 6:
File Organization & Index Structures [6L]
File & Record Concept, Placing file records on Disk, Fixed and Variable sized Records, Types of Single-Level Index (primary, secondary, clustering), Multilevel Indexes

Disk storage

Databases are stored physically as files of records, which are typically stored on magnetic disks. This chapter and the next deal with the organization of databases in storage and the techniques for accessing them efficiently using various algorithms, some of which require auxiliary data structures called indexes. These structures are often referred to as physical database file structures, and are at the physical level of the three-schema architecture described in module 1.

6.1 Introduction

The collection of data that makes up a computerized database must be stored physically on some computer storage medium. The DBMS software can then retrieve, update, and process this data as needed. Computer storage media form a storage hierarchy that includes two main categories:

■ Primary storage. This category includes storage media that can be operated on directly by the computer's central processing unit (CPU), such as the computer's main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity.

Although main memory capacities have been growing rapidly in recent years, they are still more expensive and have less storage capacity than secondary and tertiary storage devices.

■ Secondary and tertiary storage. This category includes magnetic disks, optical disks (CD-ROMs, DVDs, and other similar storage media), and tapes. Hard-disk drives are classified as secondary storage, whereas removable media such as optical disks and tapes are considered tertiary storage. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary or tertiary storage cannot be processed directly by the CPU; first it must be copied into primary storage and then processed by the CPU.

6.1.1 Memory Hierarchies and Storage Devices

In a modern computer system, data resides and is transported throughout a hierarchy of storage media. The highest-speed memory is the most expensive and is there-fore available with the least capacity. The lowest-speed memory is offline tape storage, which is essentially available in indefinite storage capacity.

At the primary storage level, the memory hierarchy includes at the most expensive end, cache memory, which is a static RAM (Random Access Memory). Cache memory is typically used by the CPU to speed up execution of program instructions using techniques such as prefetching and pipelining. The next level of primary storage is DRAM (Dynamic RAM), which provides the main work area for the CPU for keeping program instructions and data. It is popularly called main memory. The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility1 and lower speed compared with static RAM. At the secondary and tertiary storage level, the hierarchy includes magnetic disks, as well as mass storage in the form of CD-ROM (Compact Disk–Read-Only Memory) and DVD (Digital Video Disk or Digital Versatile Disk) devices, and finally tapes at the least expensive end of the hierarchy. The storage capacity is measured in kilobytes (Kbyte or 1000 bytes), megabytes (MB or 1 million bytes), gigabytes (GB or 1 billion bytes), and even terabytes (1000 GB). The word petabyte (1000 terabytes or 10**15 bytes) is now becoming relevant in the context of very large repositories of data in physics, astronomy, earth sciences, and other scientific applications.

Programs reside and execute in DRAM. Generally, large permanent databases reside on secondary storage, (magnetic disks), and portions of the database are read into and written from buffers in main memory as needed. Nowadays, personal computers and workstations have large main memories of hundreds of megabytes of RAM and DRAM, so it is becoming possible to load a large part of the database into main memory. Eight to 16 GB of main memory on a single server is becoming common-place. In some cases, entire databases can be kept in main memory (with a backup copy on magnetic disk), leading to main memory databases; these are particularly

useful in real-time applications that require extremely fast response times. An example is telephone switching applications, which store databases that contain routing and line information in main memory.
Between DRAM and magnetic disk storage, another form of memory, flash memory, is becoming common, particularly because it is nonvolatile. Flash memories are high-density, high-performance memories using EEPROM (Electrically Erasable Programmable Read-Only Memory) technology. The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over simultaneously. Flash memory cards are appearing as the data storage medium in appliances with capacities ranging from a few megabytes to a few gigabytes. These are appearing in cameras, MP3 players, cell phones, PDAs, and so on. USB (Universal Serial Bus) flash drives have become the most portable medium for carrying data between personal computers; they have a flash memory storage device integrated with a USB interface.
CD-ROM (Compact Disk – Read Only Memory) disks store data optically and are read by a laser. CD-ROMs contain prerecorded data that cannot be overwritten. WORM (Write-Once-Read-Many) disks are a form of optical storage used for achieving data; they allow data to be written once and read any number of times without the possibility of erasing. They hold about half a gigabyte of data per disk and last much longer than magnetic disks **Optical jukebox memories** use an array of CD-ROM platters, which are loaded onto drives on demand. Although optical jukeboxes have capacities in the hundreds of gigabytes, their retrieval times are in the hundreds of milliseconds, quite a bit slower than magnetic disks. This type of storage is continuing to decline because of the rapid decrease in cost and increase in capacities of magnetic disks. The DVD is another standard for optical disks allowing 4.5 to 15 GB of storage per disk. Most personal computer disk drives now read CDROM and DVD disks. Typically, drives are CD-R (Compact Disk Recordable) that can create CD-ROMs and audio CDs (Compact Disks), as well as record on DVDs.

Finally, magnetic tapes are used for archiving and backup storage of data. Tape jukeboxes—which contain a bank of tapes that are catalogued and can be automatically loaded onto tape drives—are becoming popular as tertiary storage to hold terabytes of data. For example, NASA's EOS (Earth Observation Satellite) system stores archived databases in this fashion.
Many large organizations are already finding it normal to have terabyte-sized data-bases. The term very large database can no longer be precisely defined because disk storage capacities are on the rise and costs are declining. Very soon the term may be reserved for databases containing tens of terabytes.

6.2 Secondary Storage Devices

Databases typically store large amounts of data that must persist over long periods of time, and hence is often referred to as persistent data. Parts of this data are accessed and processed repeatedly during this period. This contrasts with the notion of transient data that persist for only a limited time during program execution.
Most databases are stored permanently (or persistently) on magnetic disk secondary storage, for the following reasons:
■ Generally, databases are too large to fit entirely in main memory.
■ The circumstances that cause permanent loss of stored data arise less frequently for disk secondary storage than for primary storage. Hence, we refer to disk—and other secondary storage devices—as nonvolatile storage, whereas main memory is often called volatile storage.
■ The cost of storage per unit of data is an order of magnitude less for disk secondary storage than for primary storage.
Some of the newer technologies—such as optical disks, DVDs, and tape juke-boxes—are likely to provide viable alternatives to the use of magnetic disks. In the future, databases may therefore reside at different levels of the memory hierarchy. However, it is anticipated that magnetic disks will continue to be the primary medium of choice for large databases for years to come. Hence, it

is important to study and understand the properties and characteristics of magnetic disks and the way data files can be organized on disk in order to design effective databases with acceptable performance.
Magnetic tapes are frequently used as storage medium for backing up databases because storage on tape costs even less than storage on disk. However, access to data on tape is quite slow. Data stored on tapes is offline; that is, some intervention by an operator—or an automatic loading device—to load a tape is needed before the data becomes available. In contrast, disks are online devices that can be accessed directly at any time.
The techniques used to store large amounts of structured data on disk are important for database designers, the DBA, and implementers of a DBMS. Database designers and the DBA must know the advantages and disadvantages of each storage technique when they design, implement, and operate a database on a specific DBMS. Usually, the DBMS has several options available for organizing the data. The process of physical database design involves choosing the particular data organization techniques that best suit the given application requirements from among the options. DBMS system implementers must study data organization techniques so that they can implement them efficiently and thus provide the DBA and users of the DBMS with sufficient options.

Typical database applications need only a small portion of the database at a time for processing. Whenever a certain portion of the data is needed, it must be located on disk, copied to main memory for processing, and then rewritten to the disk if the data is changed. The data stored on disk is organized as files of records. Each record is a collection of data values that can be interpreted as facts about entities, their attributes, and their relationships. Records should be stored on disk in a manner that makes it possible to locate them efficiently when they are needed.
There are several primary file organizations, which determine how the file records are physically placed on the disk, and hence how the records can be accessed. A heap file (or unordered file) places the records on disk in no particular order by appending new records at the end of the file, whereas a sorted file (or sequential file) keeps the records ordered by the value of a particular field (called the sort key). A hashed file uses a hash function applied to a particular field (called the hash key) to determine
a record's placement on disk. Other primary file organizations, such as B-trees, use tree structures. A secondary organization or auxiliary access structure allows efficient access to file records based on alternate fields than those that have been used for the primary file organization.

6.2 Secondary Storage Devices

In this section we describe some characteristics of magnetic disk and magnetic tape storage devices. Readers who have already studied these devices may simply browse through this section.

6.2.1 Hardware Description of Disk Devices

Magnetic disks are used for storing large amounts of data. The most basic unit of data on the disk is a single bit of information. By magnetizing an area on disk in certain ways, one can make it represent a bit value of either 0 (zero) or 1 (one). To code information, bits are grouped into bytes (or characters). Byte sizes are typically 4 to 8 bits, depending on the computer and the device. We assume that one character is stored in a single byte, and we use the terms byte and character interchangeably. The capacity of a disk is the number of bytes it can store, which is usually very large. Small floppy disks used with microcomputers typically hold from 400 KB to 1.5 MB; they are rapidly going out of circulation. Hard disks for personal computers typically hold from several hundred MB up to tens of GB; and large disk packs used with servers and mainframes have capacities of hundreds of GB. Disk capacities continue to grow as technology improves. Whatever their capacity, all disks are made of magnetic material shaped as a thin circular disk, as shown in Figure 6.1(a), and protected by a plastic or acrylic cover A disk is single-sided if it stores information on one of its surfaces only and double-sided if both surfaces are used. To increase storage capacity, disks are assembled into a disk pack, as shown in Figure

6.1(b), which may include many disks and therefore many surfaces. Information is stored on a disk surface in concentric circles of small width,3 each having a distinct diameter. Each circle is called a track. In disk packs, tracks with the same diameter on the various surfaces are called a cylinder because of the shape they would form if connected in space. The concept of a cylinder is important because data stored on one cylinder can be retrieved much faster than if it were distributed among different cylinders. The number of tracks on a disk ranges from a few hundred to a few thousand, and the capacity of each track typically ranges from tens of Kbytes to 150 Kbytes. Because a track usually contains a large amount of information, it is divided into smaller blocks or sectors. The division of a track into sectors is hard-coded on the disk surface and cannot be changed. One type of sector organization, as shown in Figure 6.2(a), calls a portion of a track that subtends a fixed angle at the center a sector. Several other sector organizations are possible, one of which is to have the sectors subtend smaller angles at the center as one moves away, thus maintaining a uniform density of recording, as shown in Figure 6.2(b). A technique called ZBR (Zone Bit Recording) allows a range of cylinders to have the same number of sectors per arc. For example, cylinders 0–99 may have one sector per track, 100–199 may have two per track, and so on. Not all disks have their tracks divided into sectors.

Table 6.1: Example of Disk drive

Description	Cheetah 15K.6	Cheetah NS 10K
Model Number	ST3450856SS/FC	ST3400755FC
Height	25.4 mm	26.11 mm
Width	101.6 mm	101.85 mm
Length	146.05 mm	147 mm
Weight	0.709 kg	0.771 kg
Capacity		
Formatted Capacity	450 Gbytes	400 Gbytes
Configuration		
Number of disks (physical)	4	4
Number of heads (physical)	8	8
Performance		
Transfer Rates		
Internal Transfer Rate (min)	1051 Mb/sec	
Internal Transfer Rate (max)	2225 Mb/sec	1211 Mb/sec
Mean Time Between Failure (MTBF)		1.4 M hours
Seek Times		
Avg. Seek Time (Read)	3.4 ms (typical)	3.9 ms (typical)
Avg. Seek Time (Write)	3.9 ms (typical)	4.2 ms (typical)
Track-to-track, Seek, Read	0.2 ms (typical)	0.35 ms (typical)
Track-to-track, Seek, Write	0.4 ms (typical)	0.35 ms (typical)
Average Latency	2 ms	2.98 msec

The division of a track into equal-sized disk blocks (or pages) is set by the operating system during disk formatting (or initialization). Block size is fixed during initialization and cannot be changed dynamically. Typical disk block sizes range from 512 to 8192 bytes. A disk with hard-coded sectors often has the sectors subdivided into blocks during initialization. Blocks are separated by fixed-size interblock gaps, which include specially coded control information written during disk initialization. This information is used to determine which block on the track follows each inter block gap. Table 6.1 illustrates the specifications of typical disks used on large servers in industry. The 10K and 15K prefixes on disk names refer to the rotational speeds in rpm (revolutions per minute).

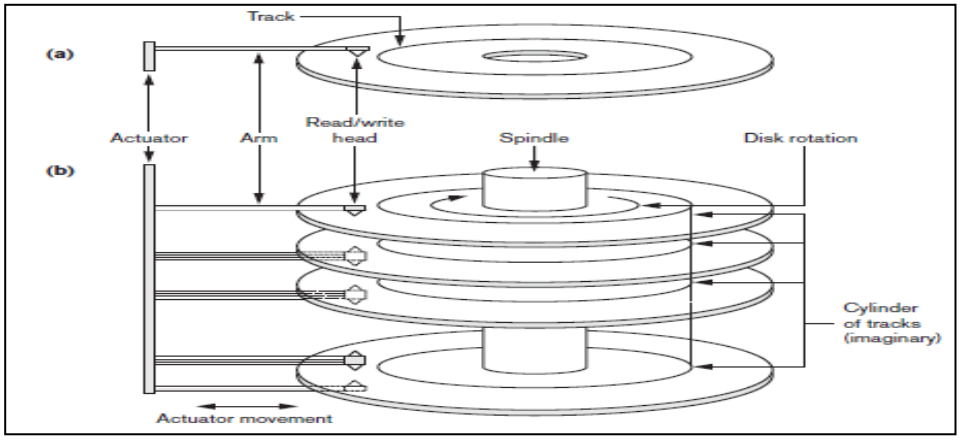


Figure 6.1: Secondary storage

There is continuous improvement in the storage capacity and transfer rates associated with disks; they are also progressively getting cheaper—currently costing only a fraction of a dollar per megabyte of disk storage. Costs are going down so rapidly that costs as low 0.025 cent/MB—which translates to $0.25/GB and $250/TB—are already here.

A disk is a random access addressable device. Transfer of data between main memory and disk takes place in units of disk blocks. The hardware address of a block—a combination of a cylinder number, track number (surface number within the cylinder on which the track is located), and block number (within the track) is supplied to the disk I/O (input/output) hardware. In many modern disk drives, a single number called LBA (Logical Block Address), which is a number between 0 and n (assuming the total capacity of the disk is n + 1 blocks), is mapped automatically to the right block by the disk drive controller. The address of a buffer—a contiguous reserved area in main storage that holds one disk block—is also provided. For a read command, the disk block is copied into the buffer; whereas for a write command, the contents of the buffer are copied into the disk block. Sometimes several contiguous blocks, called a cluster, may be transferred as a unit. In this case, the buffer size is adjusted to match the number of bytes in the cluster.

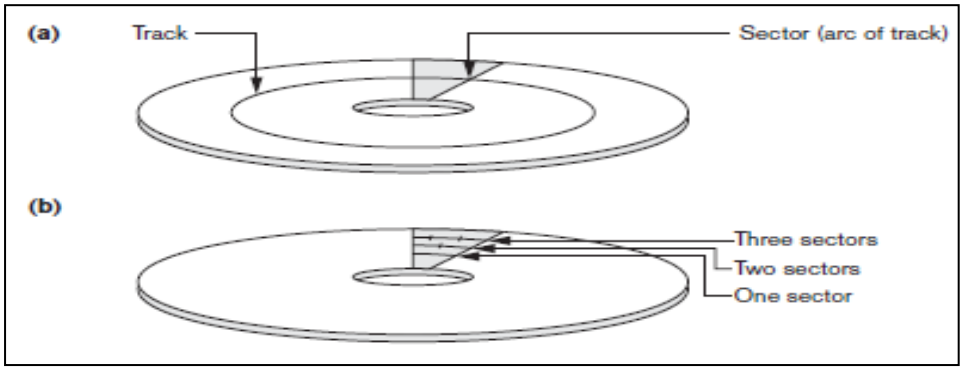


Figure 6.2: Hardware mechanism

The actual hardware mechanism that reads or writes a block is the disk read/write head, which is part of a system called a disk drive. A disk or disk pack is mounted in the disk drive, which includes a motor that rotates the disks. A read/write head includes an electronic component attached to a mechanical arm. Disk packs with multiple surfaces are controlled by several read/write heads—one for each surface, as shown in Figure 6.1(b). All arms are connected to an actuator attached to another electrical motor, which moves the read/write heads in unison and positions them precisely over the cylinder of tracks specified in a block address.

Disk drives for hard disks rotate the disk pack continuously at a constant speed (typically ranging between 5,400 and 15,000 rpm). Once the read/write head is positioned on the right track and the block specified in the block address moves under the read/write head, the electronic component of the read/write head is activated to transfer the data. Some disk units have fixed read/write heads, with as many heads as there are tracks. These are called fixed-head disks, whereas disk units with an actuator are called movable-head disks. For fixed-head disks, a track or cylinder is selected by electronically switching to the appropriate read/write head rather than by actual mechanical movement; consequently, it is much faster. However, the cost of the additional read/write heads is quite high, so fixed-head disks are not commonly used. A disk controller, typically embedded in the disk drive, controls the disk drive and interfaces it to the computer system. One of the standard interfaces used today for disk drives on PCs and workstations is called SCSI (Small Computer System Interface). The controller accepts high-level I/O commands and takes appropriate action to position the arm and causes the read/write action to

take place. To transfer a disk block, given its address, the disk controller must first mechanically position the read/write head on the correct track. The time required to do this is called the seek time. Typical seek times are 5 to 10 msec on desktops and 3 to 8 msecs on servers. Following that, there is another delay—called the rotational delay or latency—while the beginning of the desired block rotates into position under the read/write head. It depends on the rpm of the disk. For example, at 15,000 rpm, the time per rotation is 4 msec and the average rotational delay is the time per half revolution, or 2 msec. At 10,000 rpm the average rotational delay increases to 3 msec.

Finally, some additional time is needed to transfer the data; this is called the block transfer time. Hence, the total time needed to locate and transfer an arbitrary block, given its address, is the sum of the seek time, rotational delay, and block transfer time. The seek time and rotational delay are usually much larger than the block transfer time. To make the transfer of multiple blocks more efficient, it is common to transfer several consecutive blocks on the same track or cylinder. This eliminates the seek time and rotational delay for all but the first block and can result in a substantial saving of time when numerous contiguous blocks are transferred.

Usually, the disk manufacturer provides a bulk transfer rate for calculating the time required to transfer consecutive blocks. Appendix B contains a discussion of these and other disk parameters. The time needed to locate and transfer a disk block is in the order of milliseconds, usually ranging from 9 to 60 msec. For contiguous blocks, locating the first block takes from 9 to 60 msec, but transferring subsequent blocks may take only 0.4 to 2 msec each. Many search techniques take advantage of consecutive retrieval of blocks when searching for data on disk. In any case, a transfer time in the order of milliseconds is considered quite high compared with the time required to process data in main memory by current CPUs. Hence, locating data on disk is a major bottleneck in database applications. The file structures we discuss here and in Chapter 18 attempt to minimize the number of block transfers needed to locate and transfer the required data from disk to main memory. Placing “related information” on contiguous blocks is the basic goal of any storage organization on disk.

6.2.2 Magnetic Tape Storage Devices

Disks are random access secondary storage devices because an arbitrary disk block may be accessed at random once we specify its address. Magnetic tapes are sequential access devices; to access the nth block on tape, first we must scan the preceding n – 1 blocks. Data is stored on reels of high-capacity magnetic tape, somewhat similar to audiotapes or videotapes. A tape drive is required to read the data from or write the data to a tape reel. Usually, each group of bits that forms a byte is stored across the tape, and the bytes themselves are stored consecutively on the tape.

A read/write head is used to read or write data on tape. Data records on tape are also stored in blocks—although the blocks may be substantially larger than those for disks, and inter block gaps are also quite large. With typical tape densities of 1600 to 6250 bytes per inch, a typical inter block gap4 of 0.6 inch corresponds to 960 to 3750 bytes of wasted storage space. It is customary to group many records together in one block for better space utilization.

The main characteristic of a tape is its requirement that we access the data blocks in sequential order. To get to a block in the middle of a reel of tape, the tape is mounted and then scanned until the required block gets under the read/write head. For this reason, tape access can be slow and tapes are not used to store online data, except for some specialized applications. However, tapes serve a very important function—backing up the database. One reason for backup is to keep copies of disk files in case the data is lost due to a disk crash, which can happen if the disk read/write head touches the disk surface because of mechanical malfunction. For this reason, disk files are copied periodically to tape. For many online critical applications, such as airline reservation systems, to avoid any downtime, mirrored systems are used to keep three sets of identical disks—two in online operations and one as backup. Here, offline disks become a backup device. The three are rotated so that they can be switched in case there is a failure on one of the live disk drives. Tapes can also be used to store excessively large database files. Database files that are seldom used or are outdated but required for historical record keeping can be

archived on tape. Originally, half-inch reel tape drives were used for data storage employing the so-called 9 track tapes. Later, smaller 8-mm magnetic tapes (similar to those used in camcorders) that can store up to 50 GB, as well as 4-mm helical scan data cartridges and writable CDs and DVDs, became popular media for backing up data files from PCs and workstations. They are also used for storing images and system libraries.

Backing up enterprise databases so that no transaction information is lost is a major undertaking. Currently, tape libraries with slots for several hundred cartridges are used with Digital and Super digital Linear Tapes (DLTs and SDLTs) having capacities in hundreds of gigabytes that record data on linear tracks. Robotic arms are used to write on multiple cartridges in parallel using multiple tape drives with automatic labelling software to identify the backup cartridges. An example of a giant library is the SL8500 model of Sun Storage Technology that can store up to 70 petabytes (petabyte = 1000 TB) of data using up to 448 drives with a maximum throughput rate of 193.2 TB/hour. We defer the discussion of disk storage technology called RAID, and of storage area networks, network-attached storage, and iSCSI storage systems to the end of the chapter.

6.3 Buffering of Blocks

When several blocks need to be transferred from disk to main memory and all the block addresses are known, several buffers can be reserved in main memory to speed up the transfer. While one buffer is being read or written, the CPU can process data in the other buffer because an independent disk I/O processor (controller) exists that, once started, can proceed to transfer a data block between memory and disk independent of and in parallel to CPU processing. Figure 6.3 illustrates how two processes can proceed in parallel. Processes A and B are running concurrently in an interleaved fashion, whereas processes C and D are running concurrently in a parallel fashion. When a single CPU controls multiple processes, parallel execution is not possible. However, the processes can still run concurrently in an interleaved way. Buffering is most useful when processes can run concurrently in a parallel fashion, either because a separate disk I/O processor is available or because multiple CPU processors exist.

Figure 6.4 illustrates how reading and processing can proceed in parallel when the time required to process a disk block in memory is less than the time required to read the next block and fill a buffer. The CPU can start processing a block once its transfer to main memory is completed; at the same time, the disk I/O processor can be reading and transferring the next block into a different buffer. This technique is called double buffering and can also be used to read a continuous stream of blocks from disk to memory. Double buffering permits continuous reading or writing of data on consecutive disk blocks, which eliminates the seek time and rotational delay for all but the first block transfer. Moreover, data is kept ready for processing, thus reducing the waiting time in the programs.

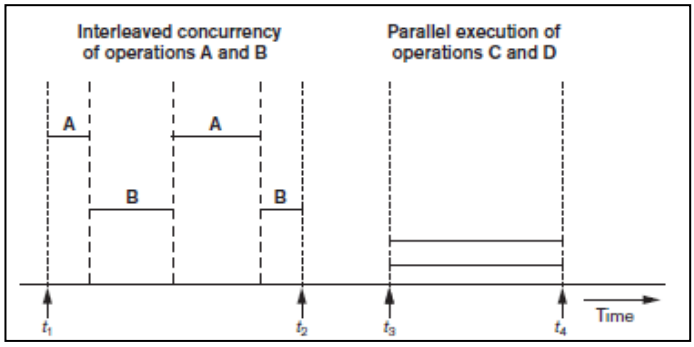


Figure 6.3: Parallel processing

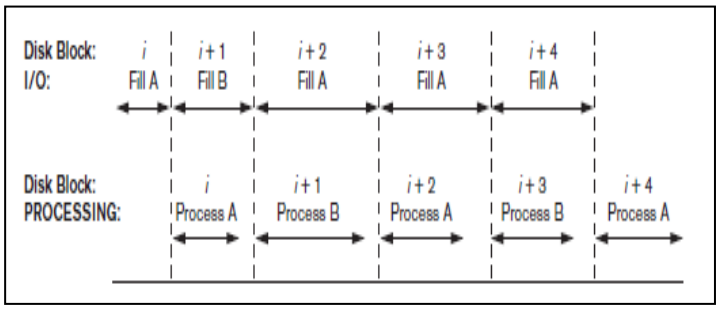


Figure 6.4: Reading and processing can proceed in parallel

6.4 Placing File Records on Disk

In this section, we define the concepts of records, record types, and files. Then we discuss techniques for placing file records on disk.

6.4.1 Records and Record Types

Data is usually stored in the form of records. Each record consists of a collection of related data values or items, where each value is formed of one or more bytes and corresponds to a particular field of the record. Records usually describe entities and their attributes. For example, an EMPLOYEE record represents an employee entity, and each field value in the record specifies some attribute of that employee, such as Name, Birth_date, Salary, or Supervisor. A collection of field names and their corresponding data types constitutes a record type or record format definition. A data type, associated with each field, specifies the types of values a field can take.

The data type of a field is usually one of the standard data types used in programming. These include numeric (integer, long integer, or floating point), string of characters (fixed-length or varying), Boolean (having 0 and 1 or TRUE and FALSE values only), and sometimes specially coded date and time data types. The number of bytes required for each data type is fixed for a given computer system. An integer may require 4 bytes, a long integer 8 bytes, a real number 4 bytes, a Boolean 1 byte, a date 10 bytes (assuming a format of YYYY-MM-DD), and a fixed-length string of k characters k bytes. Variable-length strings may require as many bytes as there are characters in each field value. For example, an EMPLOYEE record type may be defined—using the C programming language notation—as the following structure:

```
struct employee{
char name[30];
char ssn[9];
int salary;
int job_code;
char department[20];
} ;
```

In some database applications, the need may arise for storing data items that consist of large unstructured objects, which represent images, digitized video or audio streams, or free text. These are referred to as BLOBs (binary large objects). A BLOB data item is typically stored separately from its record in a pool of disk blocks, and a pointer to the BLOB is included in the record 6.4.2 Files, Fixed-Length Records, and Variable-Length Records A file is a sequence of records. In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made up of fixed-length records. If different records in the file have different sizes, the file is said to be made up of variable-length records. A file may have variable-length records for several reasons:

■ The file records are of the same record type, but one or more of the fields are of varying size (variable-length fields). For example, the Name field of EMPLOYEE can be a variable-length field.

■ The file records are of the same record type, but one or more of the fields may have multiple values for individual records; such a field is called a repeating field and a group of values for the field is often called a repeating group.
■ The file records are of the same record type, but one or more of the fields are optional; that is, they may have values for some but not all of the file records (optional fields)
■ The file contains records of different record types and hence of varying size (mixed file). This would occur if related records of different types were clustered (placed together) on disk blocks; for example, the GRADE_REPORT records of a particular student may be placed following that STUDENT's record.
The fixed-length EMPLOYEE records in Figure 6.5(a) have a record size of 71 bytes. Every record has the same fields, and field lengths are fixed, so the system can identify the starting byte position of each field relative to the starting position of the record. This facilitates locating field values by programs that access such files. Notice that it is possible to represent a file that logically should have variable-length records as a fixed-length records file. For example, in the case of optional fields, we could have every field included in every file record but store a special NULL value if no value exists for that field. For a repeating field, we could allocate as many spaces in each record as the maximum possible number of occurrences of the field. In either case, space is wasted when certain records do not have values for all the physical spaces provided in each record. Now we consider other options for formatting records of a file of variable-length records. For variable-length fields, each record has a value for each field, but we do not know the exact length of some field values. To determine the bytes within a particular record that represent each field, we can use special separator characters (such as ? or % or $)—which do not appear in any field value—to terminate variable-length fields, as shown in Figure 6.5(b), or we can store the length in bytes of the field in the record, preceding the field value.
A file of records with optional fields can be formatted in different ways. If the total number of fields for the record type is large, but the number of fields that actually appear in a typical record is small, we can include in each record a sequence of <field-name, field-value> pairs rather than just the field values. Three types of separator characters are used in Figure 6.5(c), although we could use the

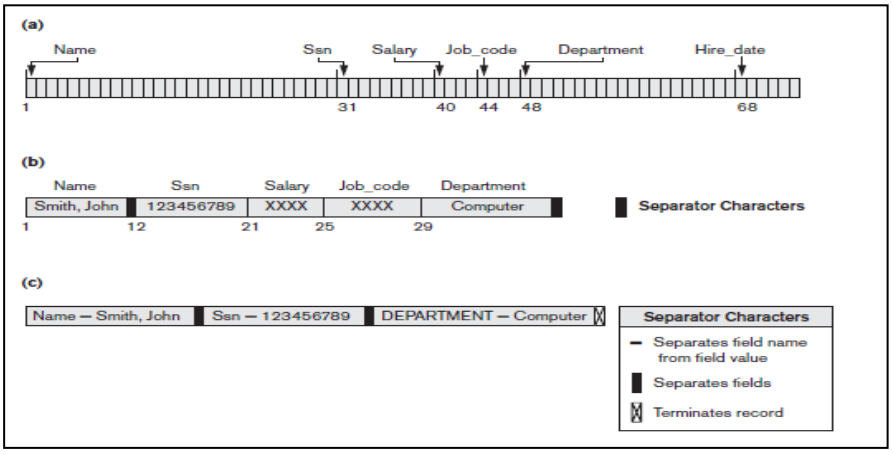


Figure 6.5: File separator

same separator character for the first two purposes—separating the field name from the field value and separating one field from the next field. A more practical option is to assign a short field type code—say, an integer number—to each field and include in each record a sequence of <field-type, field-value> pairs rather than <field-name, field-value> pairs. A repeating field needs one separator character to separate the repeating values of the field and another separator character to indicate termination of the field. Finally, for a file that includes records of different types, each record is preceded by a record type indicator. Understandably, programs that process files of variable-length records—which are usually part of the file system and hence hidden from

the typical programmers—need to be more complex than those for fixed-length records, where the starting position and size of each field are known and fixed.

6.4.3 Record Blocking and Spanned versus Unspanned Records

The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory. When the block size is larger than the record size, each block will contain numerous records, although some files may have unusually large records that cannot fit in one block. Suppose that the block size is B bytes. For a file of fixed-length records of size R bytes, with $B \geq R$, we can fit $bfr = \lfloor B/R \rfloor$ records per block, where the $\lfloor(x)\rfloor$ (floor function) rounds down the number x to an integer. The value bfr is called the blocking factor for the file. In general, R may not divide B exactly, so we have some unused space in each block equal to $B - (bfr * R)$ bytes.

To utilize this unused space, we can store part of a record on one block and the rest on another. A pointer at the end of the first block points to the block containing the remainder of the record in case it is not the next consecutive block on disk. This organization is called spanned because records can span more than one block. Whenever a record is larger than a block, we must use a spanned organization. If records are not allowed to cross block boundaries, the organization is called unspanned. This is used with fixed-length records having $B > R$ because it makes each record start at a known location in the block, simplifying record processing. For variable-length records, either a spanned or an unspanned organization can be used. If the average record is large, it is advantageous to use spanning to reduce the lost space in each block. Figure 6.6 illustrates spanned versus unspanned organization.

For variable-length records using spanned organization, each block may store a dif-ferent number of records. In this case, the blocking factor bfr represents the average number of records per block for the file. We can use bfr to calculate the number of blocks b needed for a file of r records:

$b = \lceil (r/bfr) \rceil$ blocks where the $\lceil(x)\rceil$ (ceiling function) rounds the value x up to the next integer.

6.4.4 Allocating File Blocks on Disk

There are several standard techniques for allocating the blocks of a file on disk. In contiguous allocation, the file blocks are allocated to consecutive disk blocks. This makes reading the whole file very fast using double buffering, but it makes expanding the file difficult. In linked allocation, each file block contains a pointer to the next file block. This makes it easy to expand the file but makes it slow to read the whole file. A combination of the two allocates clusters of consecutive disk blocks, and the clusters are linked. Clusters are sometimes called file segments or extents. Another possibility is to use indexed allocation, where one or more index blocks contain pointers to the actual file blocks. It is also common to use combinations of these techniques.

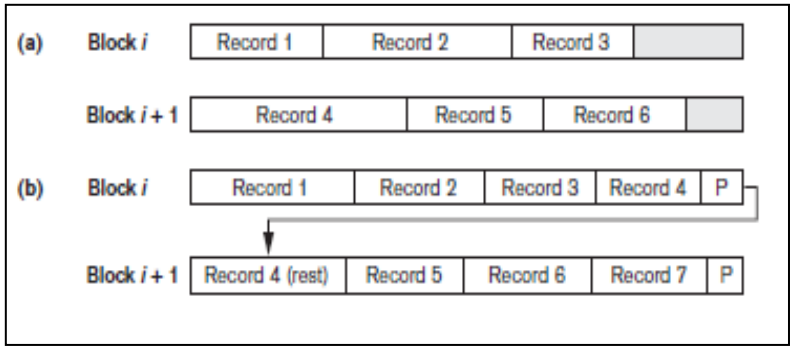


Figure 6.6: Allocating File Blocks on Disk

6.4.5 File Headers

A file header or file descriptor contains information about a file that is needed by the system programs that access the file records. The header includes information to determine the disk addresses of the file blocks as well as to record format descriptions, which may include field lengths and the order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records. To search for a record on disk, one or more blocks are copied into main memory buffers. Programs then search for the desired record or records within the buffers, using the information in the file header. If the address of the block that contains the desired record is not known, the search programs must do a linear search through the file blocks. Each file block is copied into a buffer and searched until the record is located or all the file blocks have been searched unsuccessfully. This can be very time-consuming for a large file. The goal of a good file organization is to locate the block that contains a desired record with a minimal number of block transfers.

6.5 Operations on Files

Operations on files are usually grouped into retrieval operations and update operations. The former do not change any data in the file, but only locate certain records so that their field values can be examined and processed. The latter change the file by insertion or deletion of records or by modification of field values. In either case, we may have to select one or more records for retrieval, deletion, or modification based on a selection condition (or filtering condition), which specifies criteria that the desired record or records must satisfy.

Consider an EMPLOYEE file with fields Name, Ssn, Salary, Job_code, and Department. A simple selection condition may involve an equality comparison on some field value—for example, (Ssn = ‘123456789’) or (Department = ‘Research’). More com-plex conditions can involve other types of comparison operators, such as > or ≥;an example is (Salary ≥ 30000). The general case is to have an arbitrary Boolean expression on the fields of the file as the selection condition. Search operations on files are generally based on simple selection conditions. A complex condition must be decomposed by the DBMS (or the programmer) to extract a simple condition that can be used to locate the records on disk. Each located record is then checked to determine whether it satisfies the full selection condition. For example, we may extract the simple condition (Department = ‘Research’) from the complex condition ((Salary ≥ 30000) AND (Department = ‘Research’)); each record satisfying (Department = ‘Research’) is located and then tested to see if it also satisfies (Salary ≥ 30000).

When several file records satisfy a search condition, the first record—with respect to the physical sequence of file records—is initially located and designated the current record. Subsequent search operations commence from this record and locate the next record in the file that satisfies the condition. Actual operations for locating and accessing file records vary from system to system.

Below, we present a set of representative operations. Typically, high-level programs, such as DBMS software programs, access records by using these commands, so we sometimes refer to program variables in the following descriptions:

■ Open. Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file.

■ Reset. Sets the file pointer of an open file to the beginning of the file.

■ Find (or Locate). Searches for the first record that satisfies a search condi-tion. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer, and it becomes the current record. Sometimes, different verbs are used to indicate whether the located record is to be retrieved or updated.

■ Read (or Get). Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.
■ FindNext. Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record. Various forms of FindNext (for example, Find
Next record within a current parent record, Find Next record of a given type, or Find Next record where a complex condition is met) are available in legacy DBMSs based on the hierarchical and network models.
■ Delete. Deletes the current record and (eventually) updates the file on disk to reflect the deletion.
■ Modify. Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.
■ Insert. Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.
■ Close. Completes the file access by releasing the buffers and performing any other needed cleanup operations. The preceding (except for Open and Close) are called record-at-a-time operations because each operation applies to a single record. It is possible to streamline the operations Find, FindNext, and Read into a single operation, Scan, whose description is as follows:
■ Scan. If the file has just been opened or reset, Scan returns the first record; otherwise it returns the next record. If a condition is specified with the operation, the returned record is the first or next record satisfying the condition. In database systems, additional set-at-a-time higher-level operations may be applied to a file. Examples of these are as follows:
■ FindAll. Locates all the records in the file that satisfy a search condition.
■ Find (or Locate) n. Searches for the first record that satisfies a search condition and then continues to locate the next n – 1 records satisfying the same condition. Transfers the blocks containing the n records to the main memory buffer (if not already there).
■ FindOrdered. Retrieves all the records in the file in some specified order.
■ Reorganize. Starts the reorganization process. As we shall see, some file organizations require periodic reorganization. An example is to reorder the file records by sorting them on a specified field.

At this point, it is worthwhile to note the difference between the terms file organization and access method. A file organization refers to the organization of the data of a file into records, blocks, and access structures; this includes the way records and blocks are placed on the storage medium and interlinked. An access method, on the other hand, provides a group of operations—such as those listed earlier—that can be applied to a file. In general, it is possible to apply several access methods to a file organization. Some access methods, though, can be applied only to files organized in certain ways. For example, we cannot apply an indexed access method to a file without an index (see Chapter 18). Usually, we expect to use some search conditions more than others. Some files may be static, meaning that update operations are rarely performed; other, more dynamic files may change frequently, so update operations are constantly applied to them. A successful file organization should perform as efficiently as possible the operations we expect to apply frequently to the file. For example, consider the EMPLOYEE file, as shown in Figure 6.5(a), which stores the records for current employees in a company. We expect to insert records (when employees are hired), delete records (when employees leave the company), and modify records (for exam-ple, when an employee's salary or job is changed). Deleting or modifying a record requires a selection condition to identify a particular record or set of records. Retrieving one or more records also requires a selection condition. If users expect mainly to apply a search condition based on Ssn, the designer must choose a file organization that facilitates locating a record given its Ssn value. This may involve physically ordering the records by Ssn value or

defining an index on Ssn (see Chapter 18). Suppose that a second application uses the file to generate employees' paychecks and requires that paychecks are grouped by department. For this application, it is best to order employee records by department and then by name within each department. The clustering of records into blocks and the organization of blocks on cylinders would now be different than before. However, this arrangement conflicts with ordering the records by Ssn values. If both applications are important, the designer should choose an organization that allows both operations to be done efficiently. Unfortunately, in many cases a single organization does not allow all needed operations on a file to be implemented efficiently. This requires that a compromise must be chosen that takes into account the expected importance and mix of retrieval and update operations.

In the following sections and in Chapter 18, we discuss methods for organizing records of a file on disk. Several general techniques, such as ordering, hashing, and indexing, are used to create access methods. Additionally, various general techniques for handling insertions and deletions work with many file organizations.

6.6 Files of Unordered Records (Heap Files)

In this simplest and most basic type of organization, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. Such an organization is called a heap or pile file. This organization is often used with additional access paths, such as the secondary indexes discussed in Chapter 18. It is also used to collect and store data records for future use. Inserting a new record is very efficient. The last disk block of the file is copied into a buffer, the new record is added, and the block is then rewritten back to disk. The address of the last file block is kept in the file header. However, searching for a record using any search condition involves a linear search through the file block by block—an expensive procedure. If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record. For a file of b blocks, this requires searching (b/2) blocks, on average. If no records or several records satisfy the search condition, the program must read and search all b blocks in the file. To delete a record, a program must first find its block, copy the block into a buffer, delete the record from the buffer, and finally rewrite the block back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way results in wasted storage space. Another technique used for record deletion is to have an extra byte or bit, called a deletion marker, stored with each record. A record is deleted by setting the deletion marker to a certain value. A different value for the marker indicates a valid (not deleted) record. Search programs consider only valid records in a block when conducting their search. Both of these deletion techniques require periodic reorganization of the file to reclaim the unused space of deleted records. During reorganization, the file blocks are accessed consecutively, and records are packed by removing deleted records. After such reorganization, the blocks are filled to capacity once more. Another possibility is to use the space of deleted records when inserting new records, although this requires extra bookkeeping to keep track of empty locations. We can use either spanned or unspanned organization for an unordered file, and it may be used with either fixed-length or variable-length records. Modifying a variable-length record may require deleting the old record and inserting a modified record because the modified record may not fit in its old space on disk. To read all records in order of the values of some field, we create a sorted copy of the file. Sorting is an expensive operation for a large disk file, and special techniques for external sorting are used (see Chapter 19). For a file of unordered fixed-length records using unspanned blocks and contiguous allocation, it is straightforward to access any record by its position in the file. If the file records are numbered 0, 1, 2, ..., r − 1 and the records in each block are num-bered 0, 1, ..., bfr − 1, where bfr is the blocking factor, then the ith record of the file is located in block $\lfloor (i/bfr) \rfloor$ and is the (i mod bfr)th record in that block. Such a file is often called a relative or direct file because records can easily be accessed directly by their relative positions. Accessing a record by its position does not help locate a record based on a search condition; however, it facilitates

the construction of access paths on the file, such as the indexes discussed in Chapter 18.Sometimes this organization is called a sequential file.

6.7 Files of Ordered Records (Sorted Files)

We can physically order the records of a file on disk based on the values of one of their fields—called the ordering field. This leads to an ordered or sequential file. If the ordering field is also a key field of the file—a field guaranteed to have a unique value in each record—then the field is called the ordering key for the file. Figure 6.7 shows an ordered file with Name as the ordering key field (assuming that employees have distinct names). Ordered records have some advantages over unordered files. First, reading the records in order of the ordering key values becomes extremely efficient because no sorting is required. Second, finding the next record from the current one in order of the ordering key usually requires no additional block accesses because the next record is in the same block as the current one (unless the current record is the last one in the block). Third, using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches, although it is not often used for disk files. Ordered files are blocked and stored on contiguous cylinders to minimize the seek time. A binary search for disk files can be done on the blocks rather than on the records. Suppose that the file has b blocks numbered 1, 2, ..., b; the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is K. Assuming that disk addresses of the file blocks are avail-able in the file header, the binary search can be described by Algorithm 6.1. A binary search usually accesses log2(b) blocks, whether the record is found or not—an improvement over linear searches, where, on the average, (b/2) blocks are accessed when the record is found and b blocks are accessed when the record is not found.

Algorithm 6.1. Binary Search on an Ordering Key of a Disk File
l ← 1; u ← b; (* b is the number of file blocks *)
while (u ≥ l) do
begin i ← (l + u) div 2;
read block i of the file into the buffer;
if K < (ordering key field value of the first record in block i)
then u ← i – 1
else if K > (ordering key field value of the last record in block i)
then l ← i + 1
else if the record with ordering key field value = K is in the buffer
then goto found
else goto notfound;
end;
goto notfound;

A search criterion involving the conditions >, <, ≥, and ≤ on the ordering field is quite efficient, since the physical ordering of records means that all records satisfying the condition are contiguous in the file. For example, referring to Figure 6.7, if the search criterion is (Name < ‘G’)—where < means alphabetically before—the records satisfying the search criterion are those from the beginning of the file up to the first record that has a Name value starting with the letter ‘G’.

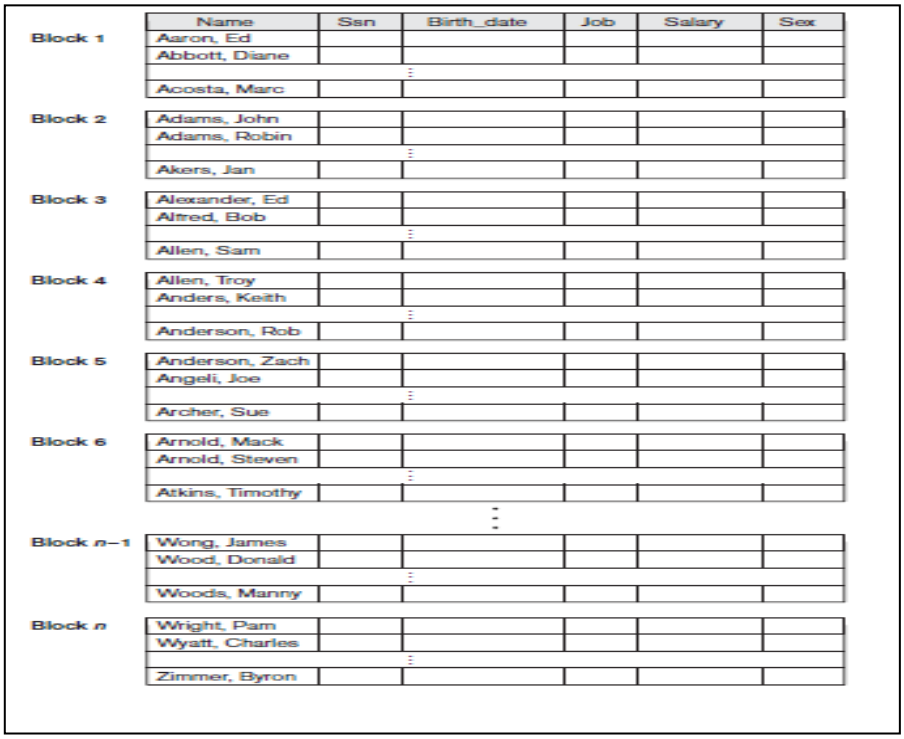


Figure 6.7: searching method

Ordering does not provide any advantages for random or ordered access of the records based on values of the other non ordering fields of the file. In these cases, we do a linear search for random access. To access the records in order based on a non ordering field, it is necessary to create another sorted copy—in a different order—of the file.

Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered. To insert a record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position. For a large file this can be very time consuming because, on the average, half the records of the file must be moved to make space for the new record. This means that half the file blocks must be read and rewritten after records are moved among them. For record deletion, the problem is less severe if deletion markers and periodic reorganization are used.

One option for making insertion more efficient is to keep some unused space in each block for new records. However, once this space is used up, the original problem resurfaces. Another frequently used method is to create a temporary unordered file called an overflow or transaction file. With this technique, the actual ordered file is called the main or master file. New records are inserted at the end of the overflow file rather than in their correct position in the main file. Periodically, the overflow file is sorted and merged with the master file during file reorganization. Insertion becomes very efficient, but at the cost of increased complexity in the search algorithm. The overflow file must be searched using a linear search if, after the binary search, the record is not found in the main file. For applications that do not require the most up-to-date information, overflow records can be ignored during a search.

Modifying a field value of a record depends on two factors: the search condition to locate the record and the field to be modified. If the search condition involves the ordering key field, we can locate the record using a binary search; otherwise we must do a linear search. A non ordering field can be modified by changing the record and rewriting it in the same physical location on disk—assuming fixed-length records. Modifying the ordering field means that the record can change its position in the file. This requires deletion of the old record followed by insertion of the modified record. Reading the file records in order of the ordering field is quite efficient if we ignore the records in overflow, since the blocks can be read consecutively using double buffering. To include the records in overflow, we must merge them in their correct positions; in

this case, first we can reorganize the file, and then read its blocks sequentially. To reorganize the file, first we sort the records in the overflow file, and then merge them with the master file. The records marked for deletion are removed during the reorganization.
Table 6.2 summarizes the average access time in block accesses to find a specific record in a file with b blocks.

Table 6.2: Average access time

Type of Organization	Access/Search Method	Average Blocks to Access a Specific Record
Heap (unordered)	Sequential scan (linear search)	$b/2$
Ordered	Sequential scan	$b/2$
Ordered	Binary search	$\log_2 b$

Ordered files are rarely used in database applications unless an additional access path, called a primary index, is used; this results in an indexed-sequential file. This further improves the random access time on the ordering key field. (We discuss indexes in Chapter 18.) If the ordering attribute is not a key, the file is called a clustered file.
6.8 Hashing Techniques

Another type of primary file organization is based on hashing, which provides very fast access to records under certain search conditions. This organization is usually called a hash file. The search condition must be an equality condition on a single field, called the hash field. In most cases, the hash field is also a key field of the file, in which case it is called the hash key. The idea behind hashing is to provide a function h, called a hash function or randomizing function, which is applied to the hash field value of a record and yields the address of the disk block in which the record is stored. A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record.
Hashing is also used as an internal search structure within a program whenever a group of records is accessed exclusively by using the value of one field. We describe the use of hashing for internal files in Section6.8.1; then we show how it is modified to store external files on disk in Section 6.8.2. In Section 6.8.3 we discuss techniques for extending hashing to dynamically growing files.

6.8.1 Internal Hashing

For internal files, hashing is typically implemented as a hash table through the use of an array of records. Suppose that the array index range is from 0 to M – 1, as shown in Figure 6.8(a); then we have M slots whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and M − 1. One common hash function is the h(K) = K mod M function, which returns the remainder of an integer hash field value K after division by M; this value is then used for the record address.
Noninteger hash field values can be transformed into integers before the mod function is applied. For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation—for example, by multiplying those code values. For a hash field whose data type is a string of 20 characters, Algorithm 6.2(a) can be used to calculate the hash address. We assume that the code function returns the numeric code of a character and that we are given a hash field value K of type K: array [1..20] of char (in Pascal) or char K[20] (in C).

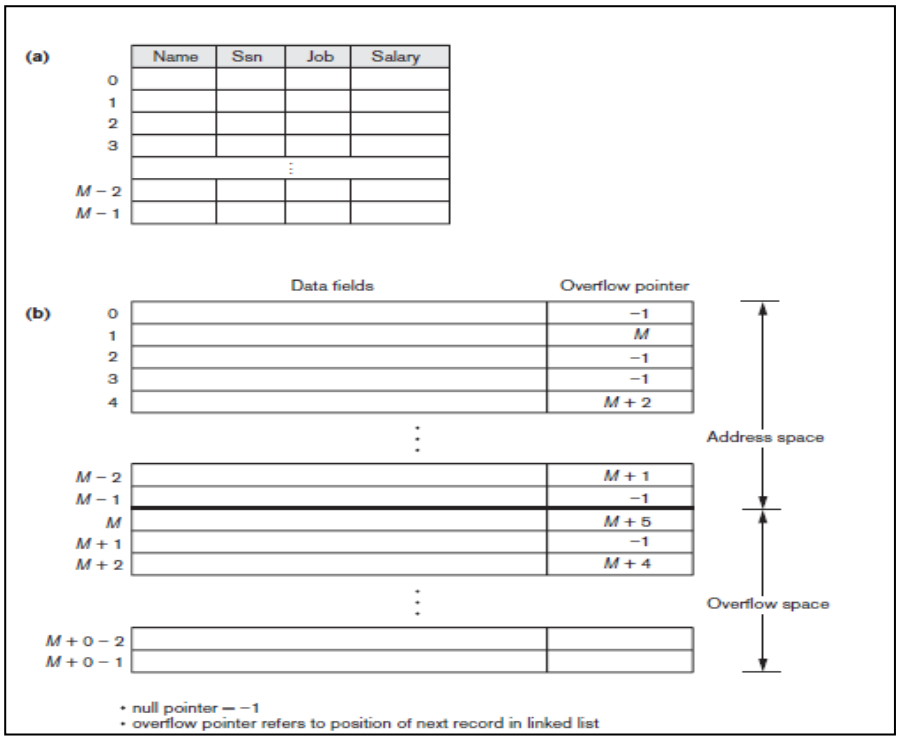


Figure 6.8: Internal hashing

Algorithm 6.2. Two simple hashing algorithms:
(a) Applying the mod hash function to a character string K.
(b) Collision resolution by open addressing.
(a) temp ← 1;
for i ← 1 to 20 do temp ← temp * code(K[i]) mod M ;
hash_address ← temp mod M;
(b) i ← hash_address(K); a ← i;
if location i is occupied
then begin i ← (i + 1) mod M;
while (i ≠ a) and location i is occupied
do i ← (i + 1) mod M;
if (i = a) then all positions are full
else new_hash_address ← i;
end;

Other hashing functions can be used. One technique, called folding, involves apply-ing an arithmetic function such as addition or a logical function such as exclusive or to different portions of the hash field value to calculate the hash address (for example, with an address space from 0 to 999 to store 1,000 keys, a 6-digit key 235469 may be folded and stored at the address: (235+964) mod 1000 = 199). Another technique involves picking some digits of the hash field value—for instance, the third,
fifth, and eighth digits—to form the hash address (for example, storing 1,000 employees with Social Security numbers of 10 digits into a hash file with 1,000 positions would give the Social Security number 301-67-8923 a hash value of 172 by this hash function).9 The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses, because the hash field space—the number of possible values a hash field can take—is usually much larger than the address space—the number of available addresses for records. The hashing function maps the hash field space to the address space.
A collision occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in

some other position, since its hash address is occupied. The process of finding another position is called collision resolution. There are numerous methods for collision resolution, including the following:

■ Open addressing. Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found. Algorithm 6.2(b) may be used for this purpose.

■ Chaining. For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. Additionally, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address is thus maintained, as shown in Figure 6.8(b).

■ Multiple hashing. The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary. Each collision resolution method requires its own algorithms for insertion, retrieval, and deletion of records. The algorithms for chaining are the simplest.

Deletion algorithms for open addressing are rather tricky. Data structures textbooks discuss internal hashing algorithms in more detail. The goal of a good hashing function is to distribute the records uniformly over the address space so as to minimize collisions while not leaving many unused locations. Simulation and analysis studies have shown that it is usually best to keep a hash table between 70 and 90 percent full so that the number of collisions remains low and we do not waste too much space. Hence, if we expect to have r records to store in the table, we should choose M locations for the address space such that (r/M) is between 0.7 and 0.9. It may also be useful to choose a prime number for M, since it has been demonstrated that this distributes the hash addresses better over the address space when the mod hashing function is used. Other hash functions may require M to be a power of 2.

6.8.2 External Hashing for Disk Files

Hashing for disk files is called external hashing. To suit the characteristics of disk storage, the target address space is made of buckets, each of which holds multiple records. A bucket is either one disk block or a cluster of contiguous disk blocks. The hashing function maps a key into a relative bucket number, rather than assigning an absolute block address to the bucket. A table maintained in the file header converts the bucket number into the corresponding disk block address, as illustrated in

Figure 6.9. The collision problem is less severe with buckets, because as many records as will fit in a bucket can hash to the same bucket without causing problems. However, we must make provisions for the case where a bucket is filled to capacity and a new record being inserted hashes to that bucket. We can use a variation of chaining in which a pointer is maintained in each bucket to a linked list of overflow records for the bucket, as shown in Figure 6.10. The pointers in the linked list should be record pointers, which include both a block address and a relative record position within the block.

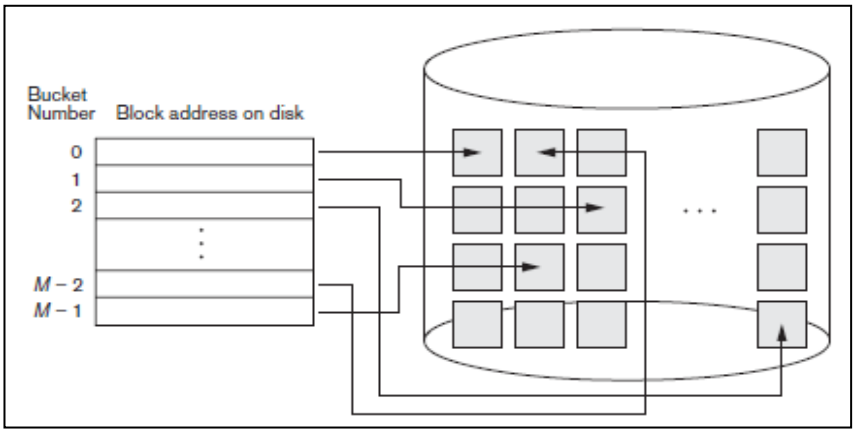


Figure 6.9: External hashing

Hashing provides the fastest possible access for retrieving an arbitrary record given the value of its hash field. Although most good hash functions do not maintain records in order of hash field values, some functions—called order preserving— do. A simple example of an order preserving hash function is to take the leftmost three digits of an invoice number field that yields a bucket address as the hash address and keep the records sorted by invoice number within each bucket. Another example is to use an integer hash key directly as an index to a relative file, if the hash key values fill up a particular interval; for example, if employee numbers in a company are assigned as 1, 2, 3, ... up to the total number of employees, we can use the identity hash function that maintains order. Unfortunately, this only works if keys are generated in order by some application. The hashing scheme described so far is called static hashing because a fixed number of buckets M is allocated. This can be a serious drawback for dynamic files. Suppose that we allocate M buckets for the address space and let m be the maximum number of records that can fit in one bucket; then at most (m * M) records will fit in the allocated space. If the number of records turns out to be substantially fewer than (m * M), we are left with a lot of unused space. On the other hand, if the number of records increases to substantially more than (m * M), numerous collisions will result and retrieval will be slowed down because of the long lists of overflow records. In either case, we may have to change the number of blocks M allocated and then use a new hashing function (based on the new value of M) to redistribute the records. These reorganizations can be quite time-consuming for large files. Newer dynamic file organizations based on hashing allow the number of buckets to vary dynamically with only localized reorganization (see Section 6.8.3).

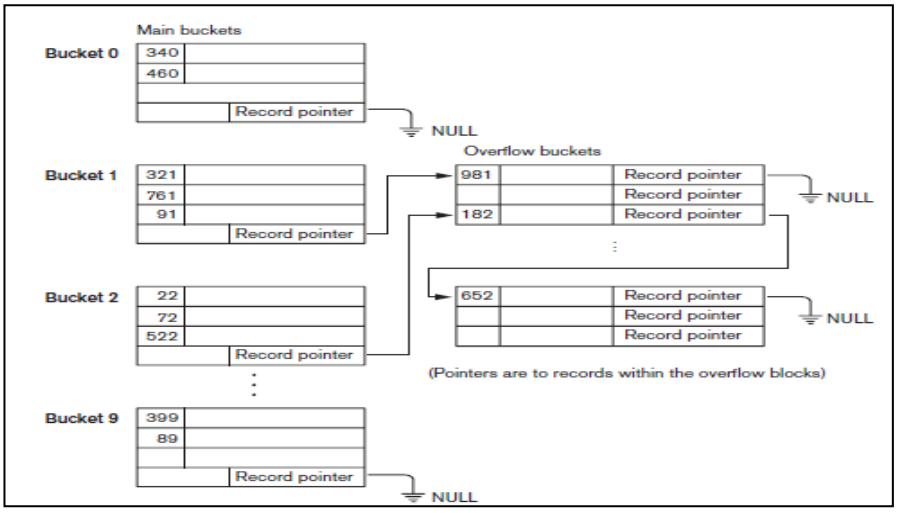


Figure 6.10: Hashing example

When using external hashing, searching for a record given a value of some field other than the hash field is as expensive as in the case of an unordered file. Record deletion can be implemented by removing the record from its bucket. If the bucket has an overflow chain, we can move one of the overflow records into the bucket to replace the deleted record. If the record to be deleted is already in overflow, we simply remove it from the linked list. Notice that removing an overflow record implies that we should keep track of empty positions in overflow. This is done easily by maintaining a linked list of unused overflow locations.

Modifying a specific record's field value depends on two factors: the search condition to locate that specific record and the field to be modified. If the search condition is an equality comparison on the hash field, we can locate the record efficiently by using the hashing function; otherwise, we must do a linear search. A nonhash field can be modified by changing the record and

rewriting it in the same bucket. Modifying the hash field means that the record can move to another bucket, which requires deletion of the old record followed by insertion of the modified record.

6.8.3 Hashing Techniques That Allow Dynamic File Expansion

A major drawback of the static hashing scheme just discussed is that the hash address space is fixed. Hence, it is difficult to expand or shrink the file dynamically. The schemes described in this section attempt to remedy this situation. The first scheme—extendible hashing—stores an access structure in addition to the file, and hence is somewhat similar to indexing (see Chapter 18). The main difference is that the access structure is based on the values that result after application of the hash function to the search field. In indexing, the access structure is based on the values of the search field itself. The second technique, called linear hashing, does not require additional access structures. Another scheme, called dynamic hashing, uses an access structure based on binary tree data structures. These hashing schemes take advantage of the fact that the result of applying a hashing function is a nonnegative integer and hence can be represented as a binary number. The access structure is built on the binary representation of the hashing function result, which is a string of bits. We call this the hash value of a record. Records are distributed among buckets based on the values of the leading bits in their hash values.

Extendible Hashing. In extendible hashing, a type of directory—an array of 2d bucket addresses—is maintained, where d is called the global depth of the directory. The integer value corresponding to the first (high-order) d bits of a hash value is used as an index to the array to determine a directory entry, and the address in that entry determines the bucket in which the corresponding records are stored. However, there does not have to be a distinct bucket for each of the 2d directory locations. Several directory locations with the same first dbits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket. A local depth d—stored with each bucket—specifies the number of bits on which the bucket contents are based. Figure 6.11 shows a directory with global depth d = 3. The value of d can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array. Doubling is needed if a bucket, whose local depth d is equal to the global depth d, overflows. Halving occurs if d >d for all the buckets after some deletions occur. Most record retrievals require two block accesses—one to the directory and the other to the bucket.

To illustrate bucket splitting, suppose that a new inserted record causes overflow in the bucket whose hash values start with 01—the third bucket in Figure 6.11. The records will be distributed between two buckets: the first contains all records whose hash values start with 010, and the second all those whose hash values start with 011. Now the two directory locations for 010 and 011 point to the two new distinct buckets. Before the split, they pointed to the same bucket. The local depth d of the two new buckets is 3, which is one more than the local depth of the old bucket. If a bucket that overflows and is split used to have a local depth

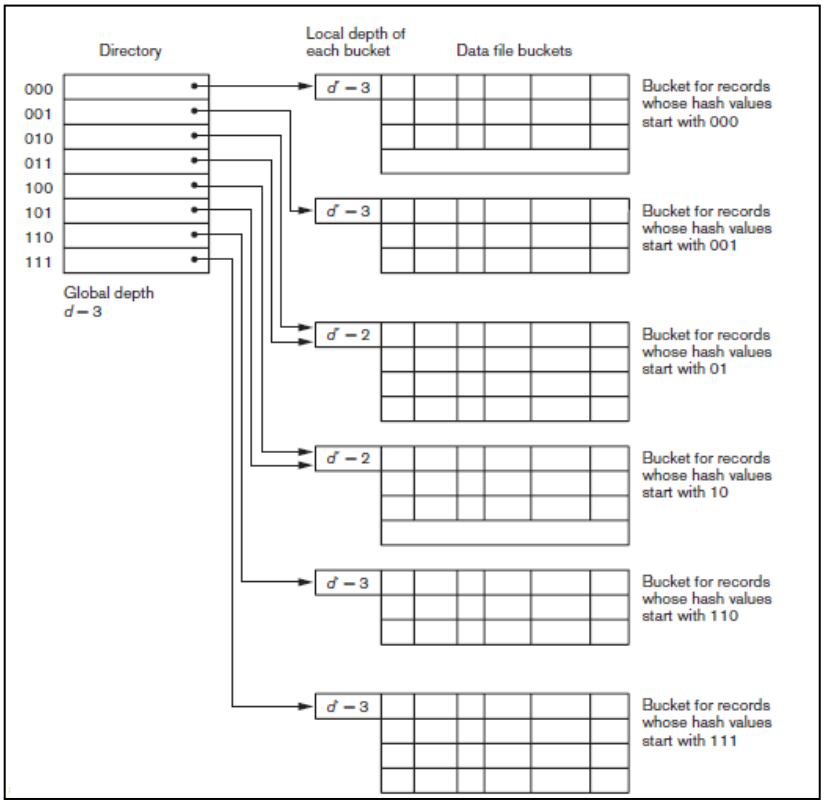


Figure 6.11: Extensible hashing

the size of the directory must now be doubled so that we can use an extra bit to distinguish the two new buckets. For example, if the bucket for records whose hash values start with 111 in Figure 6.11 overflows, the two new buckets need a directory with global depth d = 4, because the two buckets are now labelled 1110 and 1111, and hence their local depths are both 4. The directory size is hence doubled, and each of the other original locations in the directory s also split into two locations, both of which have the same pointer value as did the original location.
The main advantage of extendible hashing that makes it attractive is that the performance of the file does not degrade as the file grows, as opposed to static external hashing where collisions increase and the corresponding chaining effectively increases the average number of accesses per key. Additionally, no space is allocated in extendible hashing for future growth, but additional buckets can be allocated dynamically as needed. The space overhead for the directory table is negligible. The maximum directory size is 2k, where k is the number of bits in the hash value.
Another advantage is that splitting causes minor reorganization in most cases, since only the records in one bucket are redistributed to the two new buckets. The only time reorganization is more expensive is when the directory has to be doubled (or halved). A disadvantage is that the directory must be searched before accessing the buckets themselves, resulting in two block accesses instead of one in static hashing. This performance penalty is considered minor and thus the scheme is considered quite desirable for dynamic files.

Dynamic Hashing. A precursor to extendible hashing was dynamic hashing, in which the addresses of the buckets were either the n high-order bits or n − 1 high- order bits, depending on the total number of keys belonging to the respective bucket. The eventual storage of records in buckets for dynamic hashing is somewhat similar to extendible hashing. The major difference is in the organization of the directory. Whereas extendible hashing uses the notion of global depth (high-order d bits) for the flat directory and then combines adjacent collapsible buckets into a

bucket of local depth d − 1, dynamic hashing maintains a tree-structured directory with two types of nodes:
■ Internal nodes that have two pointers—the left pointer corresponding to the 0 bit (in the hashed address) and a right pointer corresponding to the 1 bit.
■ Leaf nodes—these hold a pointer to the actual bucket with records.

An example of the dynamic hashing appears in Figure 6.12. Four buckets are shown ("000", "001", "110", and "111") with high-order 3-bit addresses (corresponding to the global depth of 3), and two buckets ("01" and "10") are shown with high-order 2-bit addresses (corresponding to the local depth of 2). The latter two are the result of collapsing the "010" and "011" into "01" and collapsing "100" and "101" into "10". Note that the directory nodes are used implicitly to determine the "global" and "local" depths of buckets in dynamic hashing. The search for a record given the hashed address involves traversing the directory tree, which leads to the bucket holding that record. It is left to the reader to develop algorithms for insertion, deletion, and searching of records for the dynamic hashing scheme.

Linear Hashing. The idea behind linear hashing is to allow a hash file to expand and shrink its number of buckets dynamically without needing a directory. Suppose that the file starts with M buckets numbered 0, 1, ..., M − 1 and uses the mod hash function h(K) = K mod M; this hash function is called the initial hash function h_i. Overflow because of collisions is still needed and can be handled by maintaining individual overflow chains for each bucket. However, when a collision leads to an overflow record in any file bucket, the first bucket in the file—bucket 0—is split into two buckets: the original bucket 0 and a new bucket M at the end of the file. The records originally in bucket 0 are distributed between the two buckets based on a different hashing function $h_{i+1}(K) = K \bmod 2M$. A key property of the two hash functions hi and h_{i+1} is that any records that hashed to bucket 0 based on h_i will hash to either bucket 0 or bucket M based on h_{i+1}; this is necessary for linear hashing to work.
As further collisions lead to overflow records, additional buckets are split in the linear order 1, 2, 3, ... If enough overflows occur, all the original file buckets 0, 1, ..., M − 1 will have been split, so the file now has 2M instead of M buckets, and all buckets use the hash function h_{i+1}. Hence, the records in overflow are eventually redistributed into regular buckets, using the function h_{i+1} via a delayed split of their buckets. There is no directory; only a value n—which is initially set to 0 and is incremented by 1 whenever a split occurs—is needed to determine which buckets have been split. To retrieve a record with hash key value K, first apply the function hi to K; if $h_i(K) < n$, then apply the function h_{i+1} on K because the bucket is already split. Initially, n = 0, indicating that the function h_i applies to all buckets; n grows linearly as buckets are split.

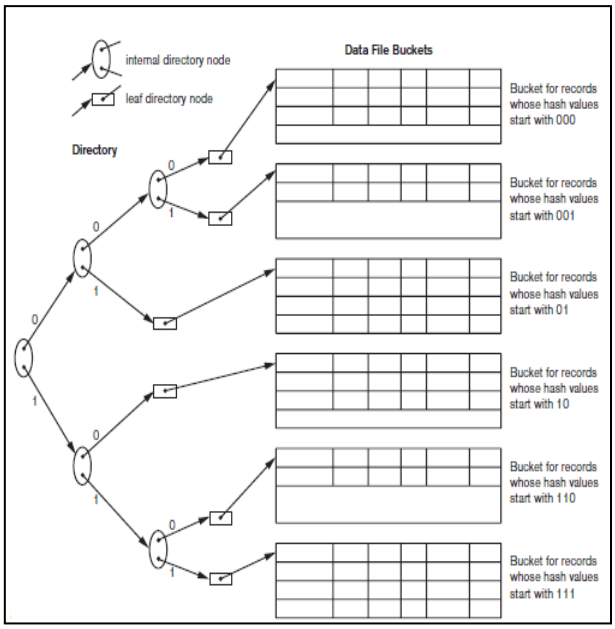


Figure 6.12: Linear hashing

When n = M after being incremented, this signifies that all the original buckets have been split and the hash function h_{i+1} applies to all records in the file. At this point, n is reset to 0 (zero), and any new collisions that cause overflow lead to the use of a new hashing function $h_{i+2}(K) = K \bmod 4M$. In general, a sequence of hashing functions $h_{i+j}(K) = K \bmod (2^jM)$ is used, where j = 0, 1, 2, ...; a new hashing function h_{i+j+1} is needed whenever all the buckets 0, 1, ..., $(2^jM) - 1$ have been split and n is reset to 0. The search for a record with hash key value K is given by Algorithm 6.3.

Splitting can be controlled by monitoring the file load factor instead of by splitting whenever an overflow occurs. In general, the file load factor l can be defined as l = r/(bfr * N), where r is the current number of file records, bfr is the maximum number of records that can fit in a bucket, and N is the current number of file buckets.

Buckets that have been split can also be recombined if the load factor of the file falls below a certain threshold. Blocks are combined linearly, and N is decremented appropriately. The file load can be used to trigger both splits and combinations; in this manner the file load can be kept within a desired range. Splits can be triggered when the load exceeds a certain threshold—say, 0.9—and combinations can be triggered when the load falls below another threshold—say, 0.7. The main advantages of linear hashing are that it maintains the load factor fairly constantly while the file grows and shrinks, and it does not require a directory.

Algorithm 6.3. The Search Procedure for Linear Hashing
if n = 0
then m ← h_j (K) (* m is the hash value of record with hash key K *)
else begin
m ← h_j (K);
if m < n then m ← h_{j+1} (K)
end;
search the bucket whose hash value is m (and its overflow, if any);

6.9 Other Primary File Organizations

6.9.1 Files of Mixed Records

The file organizations we have studied so far assume that all records of a particular file are of the same record type. The records could be of EMPLOYEEs, PROJECTs, STUDENTs, or DEPARTMENTs, but each file contains records of only one type. In most database applications, we encounter situations in which numerous types of entities are interrelated in various ways, as we saw in Chapter 7. Relationships records in various files can be represented by connecting fields.11 For example, a STUDENT record can have a connecting field Major_dept whose value gives the name of the DEPARTMENT in which the student is majoring. This Major_dept field refers to a DEPARTMENT entity, which should be represented by a record of its own in the DEPARTMENT file. If we want to retrieve field values from two related records, we must retrieve one of the records first. Then we can use its connecting field value to retrieve the related record in the other file. Hence, relationships are implemented by logical field references among the records in distinct files.

File organizations in object DBMSs, as well as legacy systems such as hierarchical and network DBMSs, often implement relationships among records as physical relationships realized by physical contiguity (or clustering) of related records or by physical pointers. These file organizations typically assign an area of the disk to hold records of more than one type so that records of different types can be physically clustered on disk. If a particular relationship is expected to be used frequently, implementing the relationship physically can increase the system's efficiency at retrieving related records. For example, if the query to retrieve a DEPARTMENT record and all records for STUDENTs majoring in that department is frequent,

it would be desirable to place each DEPARTMENT record and its cluster of STUDENT records contiguously on disk in a mixed file. The concept of physical clustering of object types is used in object DBMSs to store related objects together in a mixed file.

To distinguish the records in a mixed file, each record has—in addition to its field values—a record type field, which specifies the type of record. This is typically the first field in each record and is used by the system software to determine the type of record it is about to process. Using the catalog information, the DBMS can deter mine the fields of that record type and their sizes, in order to interpret the data values in the record.

6.9.2 B-Trees and Other Data Structures as Primary Organization

Other data structures can be used for primary file organizations. For example, if both the record size and the number of records in a file are small, some DBMSs offer the option of a B-tree data structure as the primary file organization. We will describe B-trees in Section 18.3.1, when we discuss the use of the B-tree data structure for indexing. In general, any data structure that can be adapted to the characteristics of disk devices can be used as a primary file organization for record placement on disk. Recently, column-based storage of data has been proposed as a primary method for storage of relations in relational databases. We will briefly introduce it in Chapter 18 as a possible alternative storage scheme for relational databases.

6.10 Parallelizing Disk Access Using RAID Technology

With the exponential growth in the performance and capacity of semiconductor devices and memories, faster microprocessors with larger and larger primary memories are continually becoming available. To match this growth, it is natural to expect that secondary storage technology must also take steps to keep up with processor technology in performance and reliability. A major advance in secondary storage technology is represented by the develop-ment of RAID, which originally stood for Redundant Arrays of Inexpensive Disks. More recently, the I in RAID is said to stand for Independent. The RAID idea received a very positive industry endorsement and has been developed into an elaborate set of alternative RAID architectures (RAID levels 0 through 6). We highlight the main features of the technology in this section.

The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.12 While RAM capacities have quadrupled every two to three years, disk access times are improving at less than 10 percent per year, and disk transfer rates are improving at roughly 20 percent per year. Disk capacities are indeed improving at more than 50 percent per year, but the speed and access time improvements are of a much smaller magnitude.

A second qualitative disparity exists between the ability of special microprocessors that cater to new applications involving video, audio, image, and spatial data pro- cessing (see Chapters 26 and 30 for details of these applications), with corresponding lack of fast access to large, shared data sets.

The natural solution is a large array of small independent disks acting as a single higher-performance logical disk. A concept called data striping is used, which utilizes parallelism to improve disk performance. Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk. Figure 6.13 shows a file distributed or striped over four disks. Striping improves overall I/O performance by allowing multiple I/Os to be serviced in parallel, thus providing high overall transfer rates. Data striping also accomplishes load balancing among disks. Moreover, by storing redundant information on disks using parity or some other error-correction code, reliability can be improved. In Sections 6.10.1 and 6.10.2, we discuss how RAID achieves the two important objectives of improved reliability and higher performance. Section 6.10.3 discusses RAID organizations and levels.

6.10.1 Improving Reliability with RAID

For an array of n disks, the likelihood of failure is n times as much as that for one disk. Hence, if the MTBF (Mean Time Between Failures) of a disk drive is assumed to be 200,000 hours or about 22.8 years (for the disk drive in Table 6.1 called Cheetah NS, it is 1.4 million hours), the

MTBF for a bank of 100 disk drives becomes only 2,000 hours or 83.3 days (for 1,000 Cheetah NS disks it would be 1,400 hours or 58.33 days). Keeping a single copy of data in such an array of disks will cause a significant loss of reliability. An obvious solution is to employ redundancy of data so that disk failures can be tolerated. The disadvantages are many: additional I/O operations for write, extra computation to maintain redundancy and to do recovery from errors, and additional disk capacity to store redundant information.

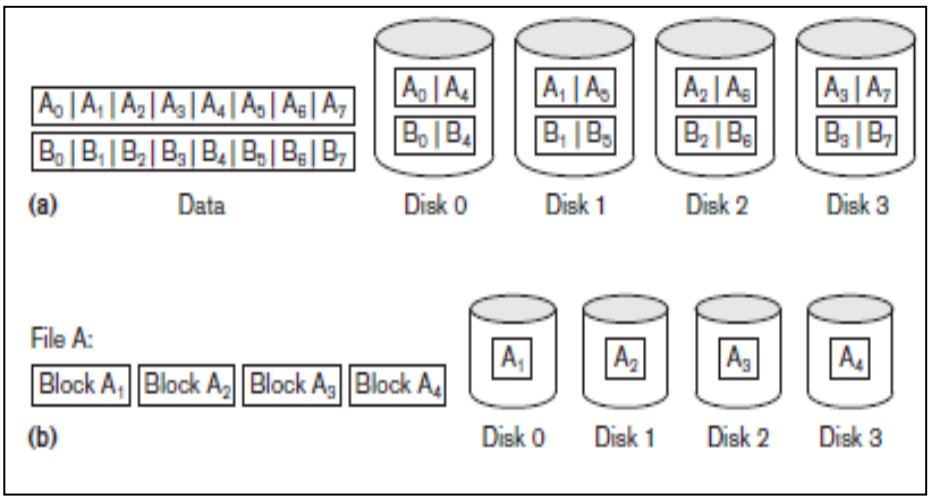


Figure 6.13: RAID Technology

One technique for introducing redundancy is called mirroring or shadowing. Data is written redundantly to two identical physical disks that are treated as one logical disk. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired. Suppose the mean time to repair is 24 hours, then the mean time to data loss of a mirrored disk system using 100 disks with MTBF of 200,000 hours each is
(200,000)2/(2 * 24) = 8.33 * 108 hours, which is 95,028 years.13 Disk mirroring also doubles the rate at which read requests are handled, since a read can go to either disk. The transfer rate of each read, however, remains the same as that for a single disk.
Another solution to the problem of reliability is to store extra information that is not normally needed but that can be used to reconstruct the lost information in case of disk failure. The incorporation of redundancy must consider two problems: selecting a technique for computing the redundant information, and selecting a method of distributing the redundant information across the disk array. The first problem is addressed by using error-correcting codes involving parity bits, or specialized codes such as Hamming codes. Under the parity scheme, a redundant disk may be considered as having the sum of all the data in the other disks. When a disk fails, the missing information can be constructed by a process similar to subtraction.
For the second problem, the two major approaches are either to store the redundant information on a small number of disks or to distribute it uniformly across all disks. The latter results in better load balancing. The different levels of RAID choose a combination of these options to implement redundancy and improve reliability.

6.10.2 Improving Performance with RAID

The disk arrays employ the technique of data striping to achieve higher transfer rates. Note that data can be read or written only one block at a time, so a typical transfer contains 512 to 8192 bytes. Disk striping may be applied at a finer granularity by breaking up a byte of data into bits and spreading the bits to different disks. Thus, bit-level data striping consists of splitting a byte of data and writing bit j to the jth disk. With 8-bit bytes, eight physical disks may be considered as one logical disk with an eightfold increase in the data transfer rate. Each disk participates in

each I/O request and the total amount of data read per request is eight times as much. Bit-level striping can be generalized to a number of disks that is either a multiple or a factor of eight. Thus, in a four-disk array, bit n goes to the disk which is (n mod 4). Figure 6.13(a) shows bit-level striping of data. The granularity of data interleaving can be higher than a bit; for example, blocks of a file can be striped across disks, giving rise to block-level striping. Figure 6.13(b) shows block-level data striping assuming the data file contains four blocks. With block-level striping, multiple independent requests that access single blocks (small requests) can be serviced in parallel by separate disks, thus decreasing the queuing time of I/O requests. Requests that access multiple blocks (large requests) can be parallelized, thus reducing their response time. In general, the more the number of disks in an array, the larger the potential performance benefit. However, assuming independent failures, the disk array of 100 disks collectively has 1/100th the reliability of a single disk. Thus, redundancy via error-correcting codes and disk mirroring is necessary to provide reliability along with high performance.

6.10.3 RAID Organizations and Levels

Different RAID organizations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information. In the initial proposal, levels 1 through 5 of RAID were proposed, and two additional levels—0 and 6—were added later. RAID level 0 uses data striping, has no redundant data, and hence has the best write performance since updates do not have to be duplicated. It splits data evenly across two or more disks. However, its read performance is not as good as RAID level 1, which uses mirrored disks. In the latter, performance improvement is possible by scheduling a read request to the disk with shortest expected seek and rotational delay. RAID level 2 uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Thus, in one particular version of this level, three redundant disks suffice for four original disks, whereas with mirroring—as in level 1—four would be required. Level 2 includes both error detection and correction, although detection is generally not required because broken disks identify themselves.

RAID level 3 uses a single parity disk relying on the disk controller to figure out which disk has failed. Levels 4 and 5 use block-level data striping, with level 5 distributing data and parity information across all disks. Figure 6.14(b) shows an illustration of RAID level 5, where parity is shown with subscript p. If one disk fails, the missing data is calculated based on the parity available from the remaining disks. Finally, RAID level 6 applies the so-called P + Q redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.

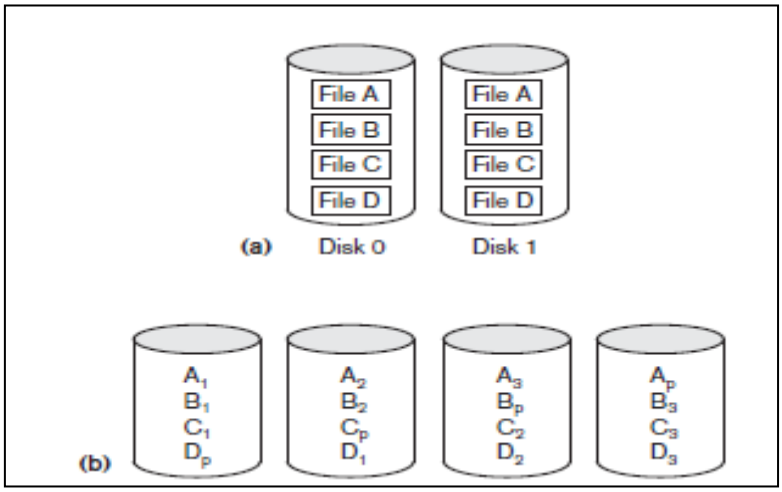


Figure 6.14: RAID Organizations

Rebuilding in case of disk failure is easiest for RAID level 1. Other levels require the reconstruction of a failed disk by reading multiple disks. Level 1 is used for critical applications such as storing logs of transactions. Levels 3 and 5 are preferred for large volume storage, with level 3 providing higher transfer rates. Most popular use of RAID technology currently uses level 0 (with striping), level 1 (with mirroring), and level 5 with an extra drive for parity. A combination of multiple RAID levels are also used – for example, 0+1 combines striping and mirroring using a minimum of four disks. Other nonstandard RAID levels include: RAID 1.5, RAID 7, RAID-DP, RAID S or Parity RAID, Matrix RAID, RAID-K, RAID-Z, RAIDn, Linux MD RAID 10, IBM ServeRAID 1E, and unRAID. A discussion of these nonstandard levels is beyond the scope of this book. Designers of a RAID setup for a given application mix have to confront many design decisions such as the level of RAID, the number of disks, the choice of parity schemes, and grouping of disks for block-level striping. Detailed performance studies on small reads and writes (referring to I/O requests for one striping unit) and large reads and writes (referring to I/O requests for one stripe unit from each disk in an error-correction group) have been performed.

6.11 New Storage Systems

In this section, we describe three recent developments in storage systems that are becoming an integral part of most enterprise's information system architectures.

6.11.1 Storage Area Networks

With the rapid growth of electronic commerce, Enterprise Resource Planning

(ERP) systems that integrate application data across organizations, and data ware- houses that keep historical aggregate information (see Chapter 29), the demand for storage has gone up substantially. For today's Internet-driven organizations, it has become necessary to move from a static fixed data center-oriented operation to a more flexible and dynamic infrastructure for their information processing requirements. The total cost of managing all data is growing so rapidly that in many instances the cost of managing server-attached storage exceeds the cost of the server itself. Furthermore, the procurement cost of storage is only a small fraction—typically, only 10 to 15 percent of the overall cost of storage management. Many users of RAID systems cannot use the capacity effectively because it has to be attached in a fixed manner to one or more servers. Therefore, most large organizations have moved to a concept called storage area networks (SANs). In a SAN, online storage peripherals are configured as nodes on a high-speed network and can be attached and detached from servers in a very flexible manner. Several companies have emerged as SAN providers and supply their own proprietary topologies. They allow storage systems to be placed at longer distances from the servers and provide different performance and connectivity options. Existing storage management applications can be ported into SAN configurations using Fiber Channel networks that encapsulate the legacy SCSI protocol. As a result, the SAN-attached devices appear as SCSI devices.

Current architectural alternatives for SAN include the following: point-to-point connections between servers and storage systems via fiber channel; use of a fiber channel switch to connect multiple RAID systems, tape libraries, and so on to servers; and the use of fiber channel hubs and switches to connect servers and storage systems in different configurations. Organizations can slowly move up from simpler topologies to more complex ones by adding servers and storage devices as needed. We do not provide further details here because they vary among SAN vendors. The main advantages claimed include:

■ Flexible many-to-many connectivity among servers and storage devices using fiber channel hubs and switches

■ Up to 10 km separation between a server and a storage system using appropriate fiber optic cables

■ Better isolation capabilities allowing non disruptive addition of new peripherals and servers SANs are growing very rapidly, but are still faced with many problems, such as combining storage options from multiple vendors and dealing with evolving standards of storage management software and hardware. Most major companies are evaluating SANs as a viable option for database storage.

6.11.2 Network-Attached Storage

With the phenomenal growth in digital data, particularly generated from multimedia and other enterprise applications, the need for high-performance storage solutions at low cost has become extremely important. Network-attached storage (NAS) devices are among the storage devices being used for this purpose. These devices are, in fact, servers that do not provide any of the common server services, but simply allow the addition of storage for file sharing. NAS devices allow vast amounts of hard-disk storage space to be added to a network and can make that space available to multiple servers without shutting them down for maintenance and upgrades. NAS devices can reside anywhere on a local area network (LAN) and may be combined in different configurations. A single hardware device, often called the NAS box or NAS head, acts as the interface between the NAS system and network clients. These NAS devices require no monitor, keyboard, or mouse. One or more disk or tape drives can be attached to many NAS systems to increase total capacity. Clients connect to the NAS head rather than to the individual storage devices. An NAS can store any data that appears in the form of files, such as e-mail boxes, Web content, remote system backups, and so on. In that sense, NAS devices are being deployed as a replacement for traditional file servers.
NAS systems strive for reliable operation and easy administration. They include built-in features such as secure authentication, or the automatic sending of e-mail alerts in case of error on the device. The NAS devices (or appliances, as some vendors refer to them) are being offered with a high degree of scalability, reliability, flexibility, and performance. Such devices typically support RAID levels 0, 1, and 5.Traditional storage area networks (SANs) differ from NAS in several ways. Specifically, SANs often utilize Fiber Channel rather than Ethernet, and a SAN often incorporates multiple network devices or end points on a self-contained or private LAN, whereas NAS relies on individual devices connected directly to the existing public LAN. Whereas Windows, UNIX, and NetWare file servers each demand specific protocol support on the client side, NAS systems claim greater operating system independence of clients.

6.11.3 iSCSI Storage Systems

A new protocol called iSCSI (Internet SCSI) has been proposed recently. It allows clients (called initiators) to send SCSI commands to SCSI storage devices on remote channels. The main advantage of iSCSI is that it does not require the special cabling needed by Fiber Channel and it can run over longer distances using existing network infrastructure. By carrying SCSI commands over IP networks, iSCSI facilitates data transfers over intranets and manages storage over long distances. It can transfer data over local area networks (LANs), wide area networks (WANs), or the Internet.
iSCSI works as follows. When a DBMS needs to access data, the operating system generates the appropriate SCSI commands and data request, which then go through encapsulation and, if necessary, encryption procedures. A packet header is added before the resulting IP packets are transmitted over an Ethernet connection. When a packet is received, it is decrypted (if it was encrypted before transmission) and disassembled, separating the SCSI commands and request. The SCSI commands go via the SCSI controller to the SCSI storage device. Because iSCSI is bidirectional, the protocol can also be used to return data in response to the original request. Cisco and IBM have marketed switches and routers based on this technology. iSCSI storage has mainly impacted small- and medium-sized businesses because of its combination of simplicity, low cost, and the functionality of iSCSI devices. It allows them not to learn the ins and outs of

Fiber Channel (FC) technology and instead benefit from their familiarity with the IP protocol and Ethernet hardware.
iSCSI implementations in the data centers of very large enterprise businesses are slow in development due to their prior investment in Fiber Channel-based SANs. iSCSI is one of two main approaches to storage data transmission over IP networks. The other method, Fiber Channel over IP (FCIP), translates Fiber Channel control codes and data into IP packets for transmission between geographically distant Fiber Channel storage area networks. This protocol, known also as Fiber Channel tunneling or storage tunneling, can only be used in conjunction with Fiber Channel technology, whereas iSCSI can run over existing Ethernet networks. The latest idea to enter the enterprise IP storage race is Fiber Channel over Ethernet (FCoE), which can be thought of as iSCSI without the IP. It uses many elements of SCSI and FC (just like iSCSI), but it does not include TCP/IP components. This promises excellent performance, especially on 10 Gigabit Ethernet (10GbE), and is relatively easy for vendors to add to their products.

6.12 Indexing Structures for Files

In this chapter we assume that a file already exists with some primary organization such as the unordered, ordered, or hashed organizations. We will describe additional auxiliary access structures called indexes, which are used to speed up the retrieval of records in response to certain search conditions. The index structures are additional files on disk that provide secondary access paths, which provide alternative ways to access the records without affecting the physical placement of records in the primary data file on disk. They enable efficient access to records based on the indexing fields that are used to construct the index. Basically, any field of the file can be used to create an index, and multiple indexes on different fields—as well as indexes on multiple fields—can be constructed on the same file. A variety of indexes are possible; each of them uses a particular data structure to speed up the search. To find a record or records in the data file based on a search condition on an indexing field, the index is searched, which leads to pointers to one or more disk blocks in the data file where the required records are located. The most prevalent types of indexes are based on ordered files (single-level indexes) and tree data structures (multilevel indexes, B+-trees). Indexes can also be constructed based on hashing or other search data structures. We also discuss indexes that are vectors of bits called bitmap indexes.

Types of Single-Level Ordered Indexes

The idea behind an ordered index is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book. We can search the book index for a certain term in the textbook to find a list of *addresses*—page numbers in this case—and use these addresses to locate the specified pages first and then *search* for the term on each specified page. The alternative, if no other guidance is given, would be to sift slowly through the whole textbook word by word to find the term we are interested in; this corresponds to doing a *linear search*, which scans the whole file. Of course, most books do have additional information, such as chapter and section titles, which help us find a term without having to search through the whole book. However, the index is the only exact indication of the pages where each term occurs in the book. For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an **indexing field** (or **indexing attribute**).1 The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are *ordered* so that we can do a *binary search* on the index. If both the data file and the index file are ordered, and since the index file is typically much smaller than the data file, searching the index using a binary search is a better option. Tree-structured multilevel indexes implement an extension of the binary search idea that reduces the search space by 2- way partitioning at each search step, thereby creating a more efficient approach that divides the search space in the file *n*-ways at each stage. There are several types of ordered indexes. A **primary index** is specified on

the *ordering key field* of an **ordered file** of records. An ordering key field is used to *physically order* the file records on disk, and every record has a *unique value* for that field. If the ordering field is not a key field—that is, if numerous records in the file can have the same value for the ordering field— another type of index, called a **clustering index**, can be used. The data file is called a **clustered file** in this latter case. Notice that a file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, *but not both*. A third type of index, called a **secondary index**, can be specified on any *non-ordering* field of a file. A data file can have several secondary indexes in addition to its primary access method. We discuss these types of single-level indexes in the next three subsections.

6.12.1 Primary Indexes

A **primary index** is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access the data records in a data file. The first field is of the same data type as the ordering key field—called the **primary key**—of the data file, and the second field is a pointer to a disk block (a block address). There is one **index entry** (or **index record**) in the index file for each *block* in the data file. Each index entry has the value of the primary key field for the *first* record in a block and a pointer to that block as its two field values. We will refer to the two field values of index entry i as $<K(i), P(i)>$. To create a primary index on the ordered file shown in Figure 6.15, we use the Name field as primary key, because that is the ordering key field of the file (assuming that each value of Name is unique). Each entry in the index has a Name value and a pointer. The first three index entries are as follows:

$<K(1) =$ (Aaron, Ed), $P(1) =$ address of block 1$>$
$<K(2) =$ (Adams, John), $P(2) =$ address of block 2$>$
$<K(3) =$ (Alexander, Ed), $P(3) =$ address of block 3$>$

Figure 6.15 illustrates this primary index. The total number of entries in the index is the same as the *number of disk blocks* in the ordered data file. The first record in each block of the data file is called the **anchor record** of the block, or simply the **block anchor**.

Indexes can also be characterized as dense or sparse. A **dense index** has an index entry for *every search key value* (and hence every record) in the data file. A **sparse** (or **nondense**) **index**, on the other hand, has index entries for only some of the search values. A sparse index has fewer entries than the number of records in the file. Thus, a primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value (or every record).

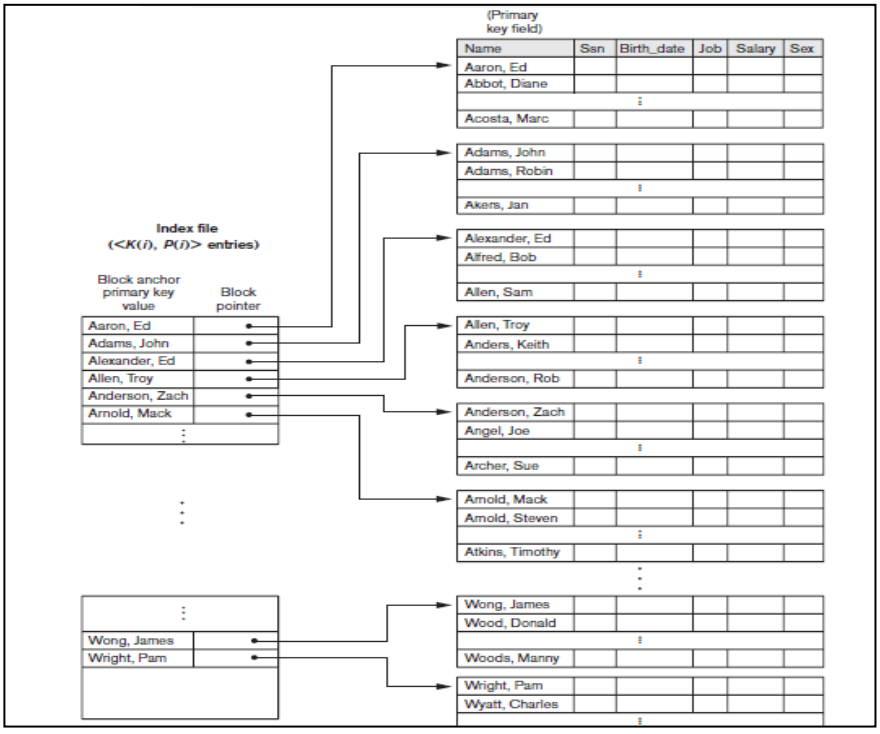


Figure 6.15: Sample Index file

The index file for a primary index occupies a much smaller space than does the data file, for two reasons. First, there are *fewer index entries* than there are records in the data file. Second, each index entry is typically *smaller in size* than a data record because it has only two fields; consequently, more index entries than data records can fit in one block. Therefore, a binary search on the index file requires fewer block accesses than a binary search on the data file. Referring to Table 6.2, note that the binary search for an ordered data file required $\log_2 b$ block accesses. But if the primary index file contains only b_i blocks, then to locate a record with a search key value requires a binary search of that index and access to the block containing that record: a total of $\log_2 b_i + 1$ accesses.

A record whose primary key value is K lies in the block whose address is $P(i)$, where $K(i) \leq K < K(i + 1)$. The ith block in the data file contains all such records because of the physical ordering of the file records on the primary key field. To retrieve a record, given the value K of its primary key field, we do a binary search on the index file to find the appropriate index entry i, and then retrieve the data file block whose address is $P(i)$. Example 1 illustrates the saving in block accesses that is attainable when a primary index is used to search for a record.

Example 1. Suppose that we have an ordered file with r = 30,000 records stored on a disk with block size B = 1024 bytes. File records are of fixed size and are unspanned, with record length R = 100 bytes. The blocking factor for the file would be $bfr = \lfloor (B/R) \rfloor = \lfloor (1024/100) \rfloor = 10$ records per block. The number of blocks needed for the file is $b = \lceil (r/bfr) \rceil = \lceil (30000/10) \rceil = 3000$ blocks. A binary search on the data file would need approximately $\lceil \log_2 b \rceil = \lceil (\log_2 3000) \rceil = 12$ block accesses. Now suppose that the ordering key field of the file is V = 9 bytes long, a block pointer is P = 6 bytes long, and we have constructed a primary index for the file. The size of each index entry is $R_i = (9 + 6) = 15$ bytes, so the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block. The total number of index entries r_i is equal to the number of blocks in the data file, which is 3000. The number of index blocks is hence $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (3000/68) \rceil = 45$ blocks. To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 45) \rceil = 6$ block accesses. To search for a record using the index, we need one additional block access to the data file for a total of 6 + 1 = 7 block accesses—an improvement over binary search on the data file, which required 12 disk block accesses. A major problem with a primary index—as with any ordered file—is insertion and deletion of records. With a primary index, the problem is compounded because if we attempt to insert a record in its correct position in the data file, we must not only move records to make space for the new record but also change some index entries, since moving records will change the *anchor records* of some blocks. Using an unordered overflow file can reduce this problem. Another possibility is to use a linked list of overflow records for each block in the data file. This is similar to the method of dealing with overflow records described with hashing. Records within each block and its overflow linked list can be sorted to improve retrieval time. Record deletion is handled using deletion markers.

6.12.2 Clustering Indexes

If file records are physically ordered on a nonkey field—which *does not* have a distinct value for each record—that field is called the **clustering field** and the data file is called a **clustered file.** We can create a different type of index, called a **clustering index**, to speed up retrieval of all the records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a *distinct value* for each record. A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a disk block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, and it contains the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field. Figure 6.16 shows an example. Notice that record insertion and deletion still cause problems because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for *each value* of the clustering field; all records with that value are placed in the block (or block cluster). This makes

insertion and deletion relatively straightforward. Figure 6.18 shows this scheme. A clustering index is another example of a *nondense* index because it has an entry for every *distinct value* of the indexing field, which is a nonkey by definition and hence has duplicate values rather than a unique value for every record in the file. There is some similarity between Figures 6.15, 6.16, and 6.17 and Figures 6.11 and 6.12. An index is somewhat similar to dynamic hashing and to the directory structures used for extendible hashing. Both are searched to find a pointer to the data block containing the desired record. A main difference is that an index search uses the values of the search field itself, whereas a hash directory search uses the binary hash value that is calculated by applying the hash function to the search field.

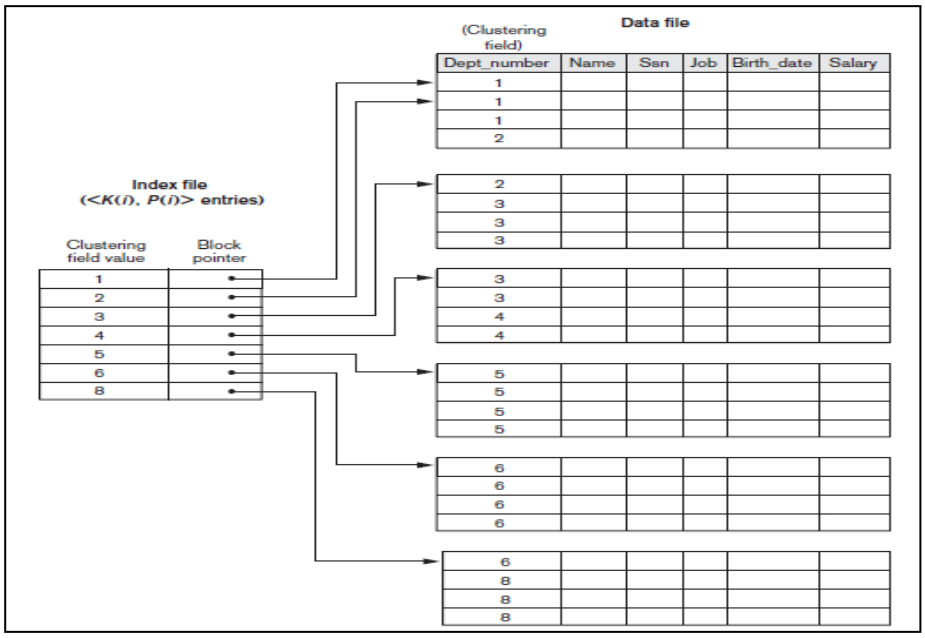


Figure 6.16: Sample clustering index file

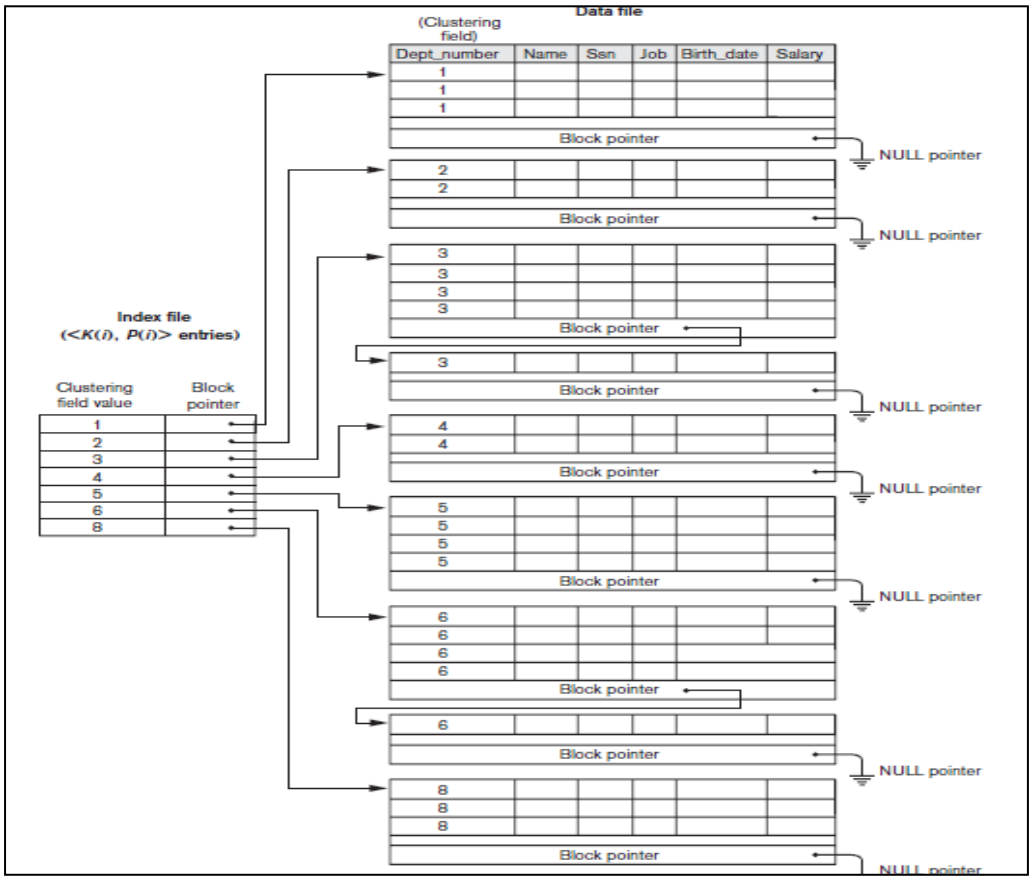


Figure 6.17: Sample clustering index with Block pointer

6.12.3 Secondary Indexes

A **secondary index** provides a secondary means of accessing a data file for which some primary access already exists. The data file records could be ordered, unordered, or hashed. The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a nonkey field with duplicate values. The index is again an ordered file with two fields. The first field is of the same data type as some *nonordering field* of the data file that is an **indexing field**. The second field is either a *block* pointer or a *record* pointer. *Many* secondary indexes (and hence, indexing fields) can be created for the same file—each represents an additional means of accessing that file based on some specific field. First we consider a secondary index access structure on a key (unique) field that has a *distinct value* for every record. Such a field is sometimes called a **secondary** key; in the relational model, this would correspond to any UNIQUE key attribute or to the primary key attribute of a table. In this case there is one index entry for *each record* in the data file, which contains the value of the field for the record and a pointer either to the block in which the record is stored or to the record itself. Hence, such an index is **dense.**

Again we refer to the two field values of index entry i as $<K(i), P(i)>$. The entries are **ordered** by value of $K(i)$, so we can perform a binary search. Because the records of the data file are *not* physically ordered by values of the secondary key field, we *cannot* use block anchors. That is why an index entry is created for each record in the data file, rather than for each block, as in the case of a primary index. Figure 6.18 illustrates a secondary index in which the pointers $P(i)$ in the index entries are *block pointers*, not record pointers. Once the appropriate disk block is transferred to a main memory buffer, a search for the desired record within the block can be carried out.

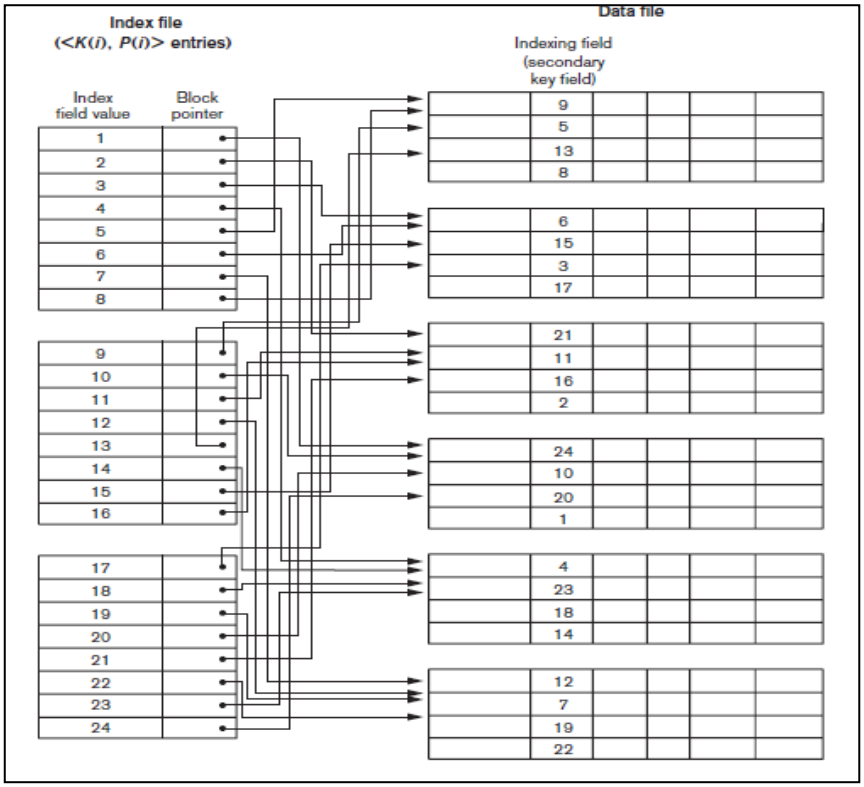


Figure 6.18: Secondary index file

A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However, the *improvement* in search time for an arbitrary record is much greater for a secondary index than for a primary index, since we would have to do a *linear search* on the data file if the secondary index did not exist. For a primary index, we could still use a binary search on the main file, even if the index did not exist. Example 2 illustrates the improvement in number of blocks accessed.

Example 2. Consider the file of Example 1 with $r = 30{,}000$ fixed-length records of size $R = 100$ bytes stored on a disk with block size $B = 1024$ bytes. The file has $b = 3000$ blocks, as calculated in Example 1. Suppose we want to search for a record with a specific value for the secondary key—a nonordering key field of the file that is $V = 9$ bytes long. Without the secondary index, to do a linear search on the file would require $b/2 = 3000/2 = 1500$ block accesses on the average. Suppose that we construct a secondary index on that *nonordering key* field of the file. As in Example 1, a block pointer is $P = 6$ bytes long, so each index entry is $R_i = (9 + 6) = 15$ bytes, and the blocking factor for the index is $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entries per block. In a dense secondary index such as this, the total number of index entries r_i is equal to the *number of records* in the data file, which is 30,000. The number of blocks needed for the index is hence $b_i = \lceil (r_i / bfr_i) \rceil = \lceil (3000/68) \rceil = 442$ blocks. A binary search on this secondary index needs $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 442) \rceil = 9$ block accesses. To search for a record using the index, we need an additional block access to the data file for a total of 9 + 1 = 10 block accesses—a vast improvement over the 1500 block accesses needed on the average for a linear search, but slightly worse than the 7 block accesses required for the primary index. This difference arose because the primary index was nondense and hence shorter, with only 45 blocks in length. We can also create a secondary index on a *nonkey, nonordering field* of a file. In this case, numerous records in the data file can have the same value for the indexing field. There are several options for implementing such an index:

- Option 1 is to include duplicate index entries with the same $K(i)$ value—one for each record. This would be a dense index.
- Option 2 is to have variable-length records for the index entries, with a repeating field for the pointer. We keep a list of pointers $<P(i, 1), ..., P(i, k)>$ in the index entry for $K(i)$—one pointer to each block that contains a record whose indexing field value equals $K(i)$. In either option 1 or option 2, the binary search algorithm on the index must be modified appropriately to account for a variable number of index entries per index key value.
- Option 3, which is more commonly used, is to keep the index entries themselves at a fixed length and have a single entry for each *index field value*, but to create *an extra level of indirection* to handle the multiple pointers. In this nondense scheme, the pointer $P(i)$ in index entry $<K(i), P(i)>$ points to a disk block, which contains a *set of record pointers*; each record pointer in that disk block points to one of the data file records with value $K(i)$ for the indexing field. If some value $K(i)$ occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used. This technique is illustrated in Figure 6.19. Retrieval via the index requires one or more additional block accesses because of the extra level, but the algorithms for searching the index and (more importantly) for inserting of new records in the data file are straightforward. In addition, retrievals on complex selection conditions may be handled by referring to the record pointers, without having to retrieve many unnecessary records from the data file. Notice that a secondary index provides a **logical ordering** on the records by the indexing field. If we access the records in order of the entries in the secondary index, we get them in order of the indexing field. The primary and clustering indexes assume that the field used for **physical ordering** of records in the file is the same as the indexing field.

6.13 Multilevel Indexes

The indexing schemes we have described thus far involve an ordered index file. A binary search is applied to the index to locate pointers to a disk block or to a record (or records) in the file

having a specific index field value. A binary search requires approximately ($\log_2 b_i$) block accesses for an index with b_i blocks because each step of the algorithm reduces the part of the index file that we continue to search by a factor of 2. This is why we take the log function to the base 2. The idea behind a **multilevel index** is to reduce the part of the index that we continue to search by bfr_i, the blocking factor for the index, which is larger than 2. Hence, the search space is reduced much faster. The value bfr_i is called the **fan-out** of the multilevel index, and we will refer to it by the symbol ***fo***. Whereas we divide the *record search space* into two halves at each step during a binary search, we divide it *n*-ways (where *n* = the fan-out) at each search step using the multilevel index. Searching a multilevel index requires approximately ($\log_{fo} b_i$) block accesses, which is a substantially smaller number than for a binary search if the fan-out is larger than 2. In most cases, the fan-out is much larger than 2. A multilevel index considers the index file, which we will now refer to as the **first** (or **base**) **level** of a

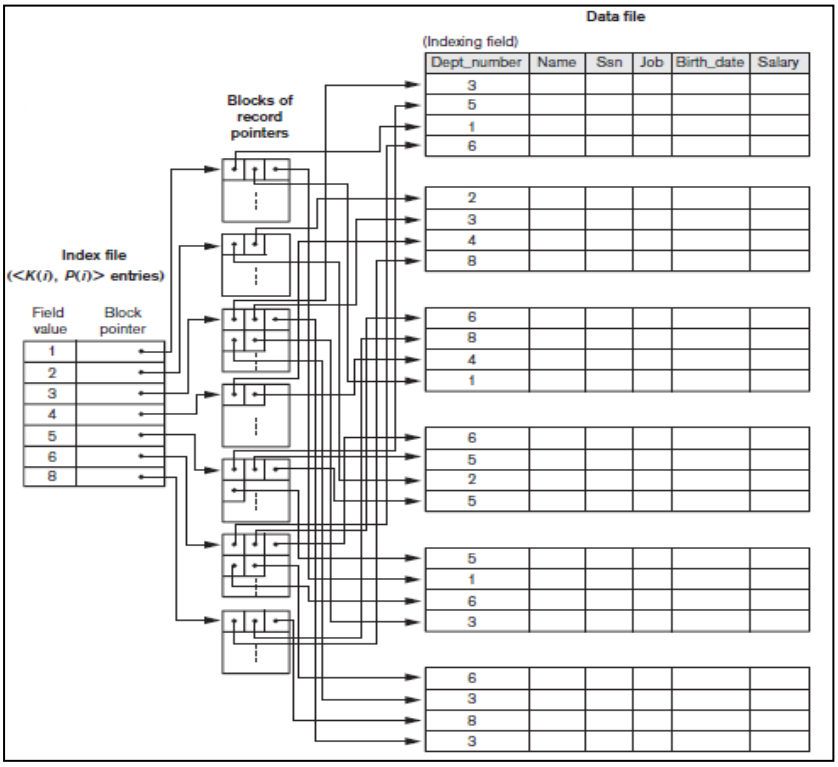


Figure 6.19: Multi level indexes

multilevel index, as an *ordered file* with a *distinct value* for each $K(i)$. Therefore, by considering the first-level index file as a sorted data file, we can create a primary index for the first level; this index to the first level is called the **second level** of the multilevel index. Because the second level is a primary index, we can use block anchors so that the second level has one entry for *each block* of the first level. The blocking factor bfr_i for the second level—and for all subsequent levels—is the same as that for the first-level index because all index entries are the same size; each has one field value and one block address. If the first level has r_1 entries, and the blocking factor—which is also the fan-out—for the index is $bfr_i = fo$, then the first level needs $\lceil (r_1/fo) \rceil$ blocks, which is therefore the number of entries r_2 needed at the second level of the index.

We can repeat this process for the second level. The **third level**, which is a primary index for the second level, has an entry for each second-level block, so the number of third-level entries is $r_3 = \lceil (r_2/fo) \rceil$ .Notice that we require a second level only if the first level needs more than one block of disk storage, and, similarly, we require a third level only if the second level needs more than one block. We can repeat the preceding process until all the entries of some index level *t* fit in a single block. This block at the *t*th level is called the **top** index level. Each level reduces the number of entries at the previous level by a factor of *fo*—the index fan-out—so we can use the formula $1 \leq (r_1/((fo)^t))$ to calculate *t*. Hence, a multilevel index with r_1 first-level entries will have

approximately t levels, where $t = \lceil (\log_{fo}(r1)) \rceil$ .When searching the index, a single disk block is retrieved at each level. Hence, t disk blocks are accessed for an index search, where t is the *number of index levels*. The multilevel scheme described here can be used on any type of index—whether it is primary, clustering, or secondary—as long as the first-level index has *distinct values for K(i) and fixed-length entries.* Figure 6.20 shows a multilevel index built over a primary index. Example 3 illustrates the improvement in number of blocks accessed when a multilevel index is used to search for a record.

Example 3. Suppose that the dense secondary index of Example 2 is converted into a multilevel index. We calculated the index blocking factor $bfr_i = 68$ index entries per block, which is also the fan-out fo for the multilevel index; the number of firstlevel blocks $b_1 = 442$ blocks was also calculated. The number of second-level blocks will be $b_2 = \lceil (b_1/fo) \rceil = \lceil (442/68) \rceil = 7$ blocks, and the number of third-level blocks will be $b_3 = \lceil (b_2/fo) \rceil = \lceil (7/68) \rceil = 1$ block. Hence, the third level is the top level of the index, and $t = 3$. To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need $t + 1 = 3 + 1 = 4$ block accesses. Compare this to Example 2, where 10 block accesses were needed when a single-level index and binary search were used.

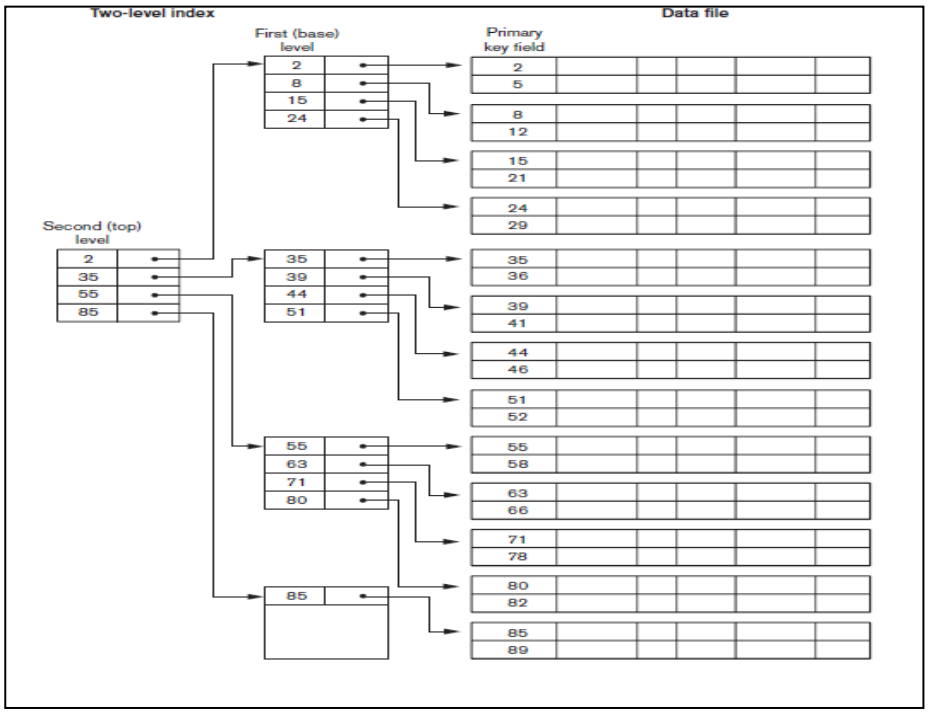


Figure 6.20: Example of secondary index

Notice that we could also have a multilevel primary index, which would be nondense. Exercise illustrates this case, where we *must* access the data block from the file before we can determine whether the record being searched for is in the file. For a dense index, this can be determined by accessing the first index level (without having to access a data block), since there is an index entry for *every* record in the file. A common file organization used in business data processing is an ordered file with a multilevel primary index on its ordering key field. Such an organization is called an **indexed sequential file** and was used in a large number of early IBM systems. IBM's **ISAM** organization incorporates a two-level index that is closely related to the organization of the disk in terms of cylinders and tracks.

The first level is a cylinder index, which has the key value of an anchor record for each cylinder of a disk pack occupied by the file and a pointer to the track index for the cylinder. The track

index has the key value of an anchor record for each track in the cylinder and a pointer to the track. The track can then be searched sequentially for the desired record or block. Insertion is handled by some form of overflow file that is merged periodically with the data file. The index is recreated during file reorganization. Algorithm 6.4 outlines the search procedure for a record in a data file that uses a nondense multilevel primary index with t levels. We refer to entry i at level j of the index as $<K_j(i), P_j(i)>$, and we search for a record whose primary key value is K. We assume that any overflow records are ignored. If the record is in the file, there must be some entry at level 1 with $K_1(i) <= K < K_1(i + 1)$ and the record will be in the block of the data file whose address is $P_1(i)$. Exercise discusses modifying the search algorithm for other types of indexes.

Algorithm 6.4. Searching a Nondense Multilevel Primary Index with t Levels
(* *We assume the index entry to be a block anchor that is the first key per block.* *)
$p \leftarrow$address of top-level block of index;
for $j \leftarrow t$ step – 1 to 1 do
begin
read the index block (at jth index level) whose address is p;
search block p for entry i such that $K_j(i) \leq K < K_j(i + 1)$
(* if $K_j(i)$
is the last entry in the block, it is sufficient to satisfy $K_j(i) \leq K$ *);
$p \leftarrow P_j(i)$ (* picks appropriate pointer at jth index level *)
end;
read the data file block whose address is p;
search block p for record with key = K;

As we have seen, a multilevel index reduces the number of blocks accessed when searching for a record, given its indexing field value. We are still faced with the problems of dealing with index insertions and deletions, because all index levels are *physically ordered files.* To retain the benefits of using multilevel indexing while reducing index insertion and deletion problems, designers adopted a multilevel index called a **dynamic multilevel index** that leaves some space in each of its blocks for inserting new entries and uses appropriate insertion/deletion algorithms for creating and deleting new index blocks when the data file grows and shrinks. It is often implemented by using data structures called B-trees and B+-trees, which we describe in the next section.

6.14 Dynamic Multilevel Indexes Using B-Trees and B+-Trees

B-trees and B+-trees are special cases of the well-known search data structure known as a **tree**. We briefly introduce the terminology used in discussing tree data structures. A **tree** is formed of **nodes**. Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent. A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node. The **level** of a node is always one more than the level of its parent, with the level of the root node being *zero*. A **subtree** of a node consists of that node and all its **descendant** nodes—its child nodes, the child nodes of its child nodes, and so on. A precise recursive definition of a subtree is that it consists of a node n and the subtrees of all the child nodes of n. Figure 6.21 illustrates a tree data structure. In this figure the root node is A, and its child nodes are B, C, and D. Nodes E, J, C, G, H, and K are leaf nodes. Since the leaf nodes are at different levels of the tree, this tree is called **unbalanced**. In next section we introduce search trees and then discuss B-trees, which can be used as dynamic multilevel indexes to guide the search for records in a data file. Btree nodes are kept between 50 and 100 percent full, and pointers to the data blocks are stored in both internal nodes and leaf nodes of the B-tree structure. In next section we discuss B+-trees, a variation of B-trees in which pointers to the data blocks of a file are stored only in leaf nodes, which can lead to fewer levels and higher capacity indexes. In the DBMSs prevalent in the market today, the common structure used for indexing is B+-trees.

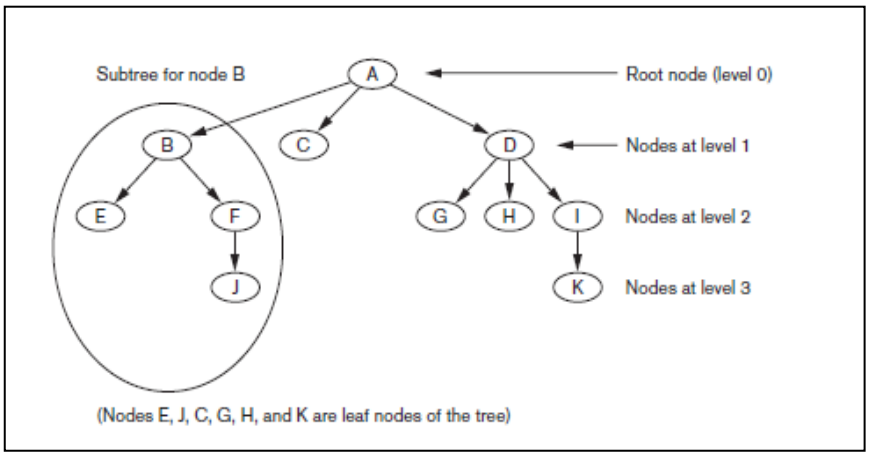


Figure 6.21: Tree data structure

6.14.1 Search Trees and B-Trees

A **search tree** is a special type of tree that is used to guide the search for a record, given the value of one of the record's fields. The multilevel indexes can be thought of as a variation of a search tree; each node in the multilevel index can have as many as *fo* pointers and *fo* key values, where *fo* is the index fan-out. The index field values in each node guide us to the next node, until we reach the data file block that contains the required records. By following a pointer, we restrict our search at each level to a subtree of the search tree and ignore all nodes not in this subtree.

Search Trees. A search tree is slightly different from a multilevel index. A **search tree of order** p is a tree such that each node contains *at most* $p - 1$ search values and p pointers in the order $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$, where $q \leq p$. Each P_i is a pointer to a child node (or a NULL pointer), and each K_i is a search value from some ordered set of values. All search values are assumed to be unique. Figure 6.22 illustrates a node in a search tree. Two constraints must hold at all times on the search tree:

1. Within each node, $K_1 < K_2 < ... < K_{q-1}$.
2. For all values X in the subtree pointed at by P_i, we have $K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 6.22).

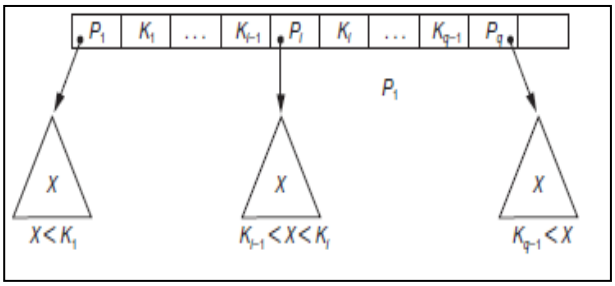


Figure 6.22: Sub tree

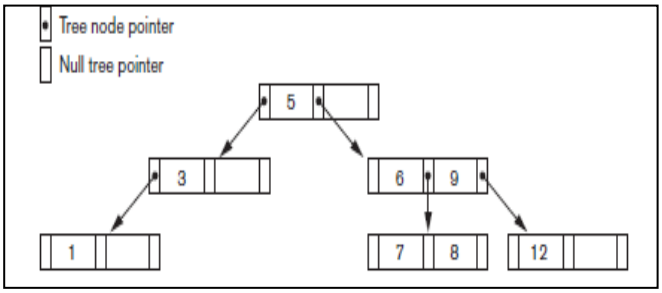


Figure 6.23: Sub tree

Whenever we search for a value X, we follow the appropriate pointer P_i according to the formulas in condition 2 above. Figure 6.23 illustrates a search tree of order $p = 3$ and integer search values. Notice that some of the pointers P_i in a node may be NULL pointers. We can use a search tree as a mechanism to search for records stored in a disk file. The values in the tree can be the values of one of the fields of the file, called the **search field** (which is the same as the index field if a multilevel index guides the search). Each key value in the tree is associated with a pointer to the record in the data file having that value. Alternatively, the pointer could be to the disk block containing that record. The search tree itself can be stored on disk by assigning each tree node to a disk block. When a new record is inserted in the file, we must update the search tree by inserting an entry in the tree containing the search field value of the new record and a pointer to the new record. Algorithms are necessary for inserting and deleting search values into and from the search tree while maintaining the preceding two constraints. In general, these algorithms do not guarantee that a search tree is **balanced**, meaning that all of its leaf nodes are at the same level. The tree in Figure 6.21 is not balanced because it has leaf nodes at levels 1, 2, and 3. The goals for balancing a search tree are as follows:

■ To guarantee that nodes are evenly distributed, so that the depth of the tree is minimized for the given set of keys and that the tree does not get skewed with some nodes being at very deep levels

■ To make the search speed uniform, so that the average time to find any random key is roughly the same

While minimizing the number of levels in the tree is one goal, another implicit goal is to make sure that the index tree does not need too much restructuring as records are inserted into and deleted from the main file. Thus we want the nodes to be as full as possible and do not want any nodes to be empty if there are too many deletions. Record deletion may leave some nodes in the tree nearly empty, thus wasting storage space and increasing the number of levels. The B-tree addresses both of these problems by specifying additional constraints on the search tree.

B-Trees. The B-tree has additional constraints that ensure that the tree is always balanced and that the space wasted by deletion, if any, never becomes excessive. The algorithms for insertion and deletion, though, become more complex in order to maintain these constraints. Nonetheless, most insertions and deletions are simple processes; they become complicated only under special circumstances—namely, whenever we attempt an insertion into a node that is already full or a deletion from a node that makes it less than half full. More formally, a **B-tree of order *p***, when used as an access structure on a *key field* to search for records in a data file, can be defined as follows:

1. Each internal node in the B-tree (Figure 6.24(a)) is of the form

$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_q>$

where $q \leq p$. Each P_i is a **tree pointer**—a pointer to another node in the Btree. Each Pr_i is a **data pointer**—a pointer to the record whose search key field value is equal to K_i (or to the data file block containing that record).

2. Within each node, $K_1 < K_2 < ... < K_{q-1}$.

3. For all search key field values X in the subtree pointed at by P_i (the ith subtree, see Figure 6.24(a)), we have:

$K_{i-1} < X < K_i$ for $1 < i < q$; $X < K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$.

4. Each node has at most p tree pointers.

5. Each node, except the root and leaf nodes, has at least $\lceil (p/2) \rceil$ tree pointers.

The root node has at least two tree pointers unless it is the only node in the tree.

6. A node with q tree pointers, $q \leq p$, has $q - 1$ search key field values (and hence has $q - 1$ data pointers).

7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their *tree pointers* P_i are NULL.

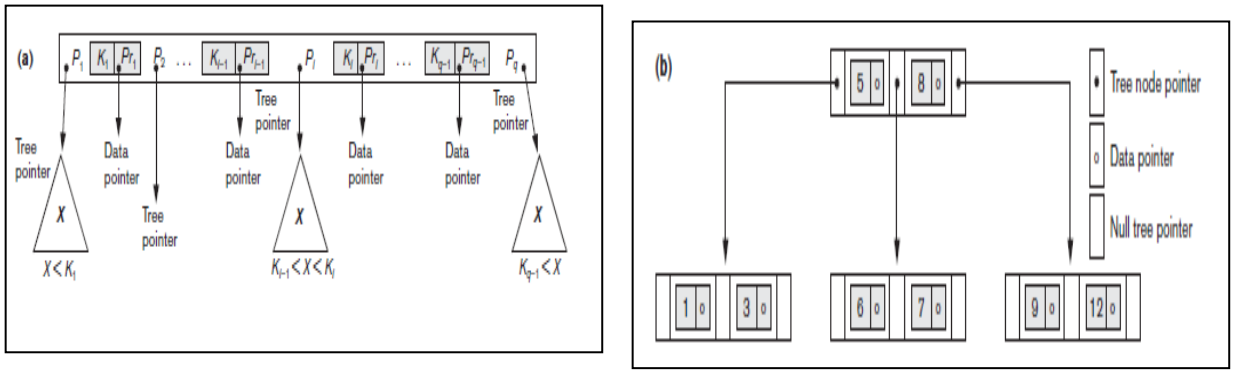


Figure 6.24: sample B tree

Figure 6.24(b) illustrates a B-tree of order $p = 3$. Notice that all search values K in the B-tree are unique because we assumed that the tree is used as an access structure on a key field. If we use a B-tree *on a nonkey field*, we must change the definition of the file pointers Pr_i to point to a block—or a cluster of blocks—that contain the pointers to the file records. This extra level of indirection is similar to option 3, for secondary indexes. A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero).

Once the root node is full with $p - 1$ search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes. If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split. We do not discuss algorithms for B trees in detail in this book, but we outline search and insertion procedures for B+-trees in the next section.

If deletion of a value causes a node to be less than half full, it is combined with its neighboring nodes, and this can also propagate all the way to the root. Hence, deletion can reduce the number of tree levels. It has been shown by analysis and simulation that, after numerous random insertions and deletions on a B-tree, the nodes are approximately 69 percent full when the number of values in the tree stabilizes. This is also true of B+-trees. If this happens, node splitting and combining will occur only rarely, so insertion and deletion become quite efficient. If the number of values grows, the tree will expand without a problem—although splitting of nodes may occur, so some insertions will take more time. Each B-tree node can have *at most p* tree pointers, $p - 1$ data pointers, and $p - 1$ search key field values (see Figure 6.24(a)).

In general, a B-tree node may contain additional information needed by the algorithms that manipulate the tree, such as the number of entries q in the node and a pointer to the parent node. Next, we illustrate how to calculate the number of blocks and levels for a B-tree.

Example 4. Suppose that the search field is a nonordering key field, and we construct a B-tree on this field with $p = 23$. Assume that each node of the B-tree is 69 percent full. Each node, on the average, will have $p * 0.69 = 23 * 0.69$ or approximately 16 pointers and, hence, 15 search key field values. The **average fan-out** $fo = 16$.We can start at the root and see how many values and pointers can exist, on the average, at each subsequent level:

Root	1 node	15 key entries	16 pointers
Level 1:	16 nodes	240 key entries	256 pointers
Level 2:	256 nodes	3840 key entries	4096 pointers
Level 3:	4096 nodes	61,440 key entries	

At each level, we calculated the number of key entries by multiplying the total number of pointers at the previous level by 15, the average number of entries in each node. Hence, for the

given block size, pointer size, and search key field size, a two level B-tree holds 3840 + 240 + 15 = 4095 entries on the average; a three-level B-tree holds 65,535 entries on the average.

B-trees are sometimes used as **primary file organizations**. In this case, *whole records* are stored within the B-tree nodes rather than just the <search key, record pointer> entries. This works well for files with a relatively *small number of records* and a *small record size.* Otherwise, the fan-out and the number of levels become too great to permit efficient access.

In summary, B-trees provide a multilevel access structure that is a balanced tree structure in which each node is at least half full. Each node in a B-tree of order p can have at most $p - 1$ search values.

6.14.2 B+-Trees

Most implementations of a dynamic multilevel index use a variation of the B-tree data structure called a **B+-tree**. In a B-tree, every value of the search field appears once at some level in the tree, along with a data pointer. In a B+-tree, data pointers are stored *only at the leaf nodes* of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes. The leaf nodes have an entry for *every* value of the search field, along with a data pointer to the record (or to the block that contains this record) if the search field is a key field. For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection. The leaf nodes of the B+-tree are usually linked to provide ordered access on the search field to the records. These leaf nodes are similar to the first (base) level of an index. Internal nodes of the B+-tree correspond to the other levels of a multilevel index. Some search field values from the leaf nodes are *repeated* in the internal nodes of the B+-tree to guide the search. The structure of the *internal nodes* of a B+- tree of order p (Figure 6.25(a)) is as follows:

1. Each internal node is of the form
<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q> where $q \leq p$ and each P_i is a **tree pointer**.
2. Within each internal node, $K_1 < K_2 < ... < K_{q-1}$.
3. For all search field values X in the subtree pointed at by P_i, we have $K_{i-1} < X \leq K_i$ for $1 < i < q$; $X \leq K_i$ for $i = 1$; and $K_{i-1} < X$ for $i = q$ (see Figure 6.25(a)).
4. Each internal node has at most p tree pointers.
5. Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
6. An internal node with q pointers, $q \leq p$, has $q - 1$ search field values.

The structure of the *leaf nodes* of a B+-tree of order p (Figure 6.25(b)) is as follows:

1. Each leaf node is of the form
<<K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{next}> where $q \leq p$, each Pr_i is a data pointer, and P_{next} points to the next *leaf node* of the B+-tree.
2. Within each leaf node, $K_1 \leq K_2 ... , K_{q-1}$, $q \leq p$.
3. Each Pr_i is a **data pointer** that points to the record whose search field value is K_i or to a file block containing the record (or to a block of record pointers that point to records whose search field value is K_i if the search field is not a key).
4. Each leaf node has at least $\lceil (p/2) \rceil$ values.
5. All leaf nodes are at the same level.

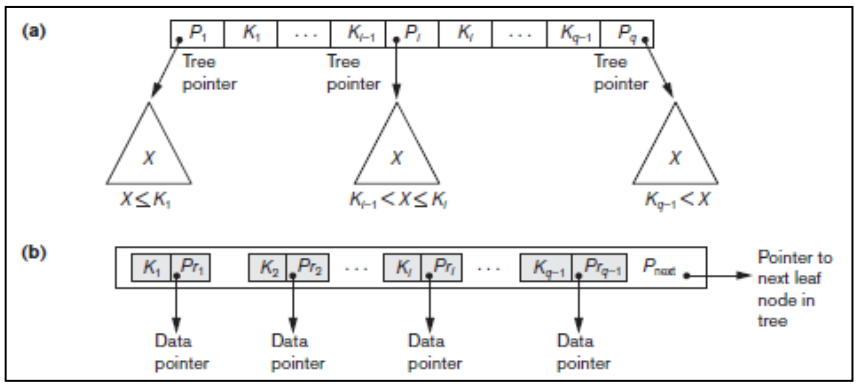


Figure 6.25: Sample B+ tree

The pointers in internal nodes are *tree pointers* to blocks that are tree nodes, whereas the pointers in leaf nodes are *data pointers* to the data file records or blocks—except for the P_{next} pointer, which is a tree pointer to the next leaf node. By starting at the leftmost leaf node, it is possible to traverse leaf nodes as a linked list, using the P_{next} pointers. This provides ordered access to the data records on the indexing field. A $P_{previous}$ pointer can also be included. For a B+-tree on a nonkey field, an extra level of indirection is needed similar to the one shown in Figure 6.19, so the P_r pointers are block pointers to blocks that contain a set of record pointers to the actual records in the data file. Because entries in the *internal nodes* of a B+-tree include search values and tree pointers without any data pointers, more entries can be packed into an internal node of a B+-tree than for a similar B-tree. Thus, for the same block (node) size, the order p will be larger for the B+-tree than for the B-tree, as we illustrate in Example 5. This can lead to fewer B+-tree levels, improving search time. Because the structures for internal and for leaf nodes of a B+-tree are different, the order p can be different. We will use p to denote the order for *internal nodes* and p_{leaf} to denote the order for *leaf nodes*, which we define as being the maximum number of data pointers in a leaf node.

Example 5. To calculate the order p of a B+-tree, suppose that the search key field is $V = 9$ bytes long, the block size is $B = 512$ bytes, a record pointer is $P_r = 7$ bytes, and a block pointer is $P = 6$ bytes. An internal node of the B+-tree can have up to p tree pointers and $p – 1$ search field values; these must fit into a single block. Hence, we have:

$(p * P) + ((p – 1) * V) \leq B$

$(P * 6) + ((P – 1) * 9) \leq 512$

$(15 * p) \leq 521$

We can choose p to be the largest value satisfying the above inequality, which gives $p = 34$. This is larger than the value of 23 for the B-tree (it is left to the reader to compute the order of the B-tree assuming same size pointers), resulting in a larger fan-out and more entries in each internal node of a B+-tree than in the corresponding B-tree. The leaf nodes of the B+-tree will have the same number of values and pointers, except that the pointers are data pointers and a next pointer. Hence, the order p_{leaf} for the leaf nodes can be calculated as follows:

$(p_{leaf} * (P_r + V)) + P \leq B$

$(p_{leaf} * (7 + 9)) + 6 \leq 512$

$(16 * p_{leaf}) \leq 506$

It follows that each leaf node can hold up to $p_{leaf} = 31$ key value/data pointer combinations, assuming that the data pointers are record pointers. As with the B-tree, we may need additional information—to implement the insertion and deletion algorithms—in each node. This information can include the type of node (internal or leaf), the number of current entries q in the node, and pointers to the parent and sibling nodes. Hence, before we do the above calculations for p and p_{leaf}, we should reduce the block size by the amount of space needed for all such information. The next example illustrates how we can calculate the number of entries in a B+-tree.

Example 6. Suppose that we construct a B+-tree on the field in Example 5. To calculate the approximate number of entries in the B+-tree, we assume that each node is 69 percent full. On the average, each internal node will have 34 * 0.69 or approximately 23 pointers, and hence 22 values. Each leaf node, on the average, will hold $0.69 * p_{leaf} = 0.69 * 31$ or approximately 21 data record pointers. A B+-tree will have the following average number of entries at each level:

Root	1 node	22 key entries	23 pointers
Level 1:	23 nodes	506 key entries	529 pointers
Level 2:	529 nodes	11638 key entries	12167 pointers
Leaf Level:	12167 nodes	255,507 data record pointers	

For the block size, pointer size, and search field size given above, a three-level B+- tree holds up to 255,507 record pointers, with the average 69 percent occupancy of nodes. Compare this to the 65,535 entries for the corresponding B-tree in Example 4. This is the main reason that B+-trees are preferred to B-trees as indexes to database files.

Search, Insertion, and Deletion with B+-Trees. Algorithm 6.5 outlines the procedure using the B+-tree as the access structure to search for a record. Algorithm 6.6 illustrates the procedure for inserting a record in a file with a B+-tree access structure. These algorithms assume the existence of a key search field, and they must be modified appropriately for the case of a B+-tree on a nonkey field. We illustrate insertion and deletion with an example.

Algorithm 6.5. Searching for a Record with Search Key Field Value *K*, Using a B+-tree

n ←block containing root node of B+-tree;
read block *n*;
while (*n* is not a leaf node of the B+-tree) do
begin
q ←number of tree pointers in node *n*;
if $K \leq n.K_1$ (*$n.K_i$ refers to the *i*th search field value in node *n**)
then $n \leftarrow n.P_1$ (*$n.P_i$ refers to the *i*th tree pointer in node *n**)
else if $K > n.K_{q-1}$
then $n \leftarrow n.P_q$
else **begin**
search node *n* for an entry *i* such that $n.K_{i-1} < K \leq n.K_i$;
$n \leftarrow n.P_i$
end;
read block *n*
end;
search block *n* for entry (K_i, Pr_i) with $K = K_i$; (* search leaf node *)
if found
then read data file block with address Pr_i and retrieve record
else the record with search field value *K* is not in the data file;

Algorithm 6.6. Inserting a Record with Search Key Field Value *K* in a B+-tree of Order *p*

n ←block containing root node of B+-tree;
read block *n*; set stack *S* to empty;
while (*n* is not a leaf node of the B+-tree) do
begin
push address of *n* on stack *S*;
(*stack *S* holds parent nodes that are needed in case of split*)
q ←number of tree pointers in node *n*;
if $K \leq n.K_1$ (*$n.K_i$ refers to the *i*th search field value in node *n**)then $n \leftarrow n.P_1$ (*$n.P_i$ refers to the *i*th tree pointer in node *n**)
else if $K > n.K_{q-1}$
then $n \leftarrow n.P_q$
else **begin**
search node *n* for an entry *i* such that $n.K_{i-1} < K\ n.K_i$;
$n \leftarrow n.P_i$
end;
read block *n*
end;
search block *n* for entry (K_i,Pr_i) with $K = K_i$; (*search leaf node *n**)
if found
then record already in file; cannot insert
else (*insert entry in B+-tree to point to record*)
begin
create entry (*K*, *Pr*) where *Pr* points to the new record;

if leaf node *n* is not full
then insert entry (*K*, *Pr*) in correct position in leaf node *n*
else **begin** (*leaf node *n* is full with p_{leaf} record pointers; is split*)
copy *n* to *temp* (**temp* is an oversize leaf node to hold extra entries*);
insert entry (*K*, *Pr*) in *temp* in correct position;
(**temp* now holds p_{leaf} + 1 entries of the form (K_i, Pr_i)*)
new ←a new empty leaf node for the tree; *new*.P_{next} ←*n*.P_{next} ;
j ← ⌈(p_{leaf} + 1)/2⌉ ;
n ←first *j* entries in *temp* (up to entry (K_j, Pr_j)); *n*.P_{next} ←*new*;
new ←remaining entries in *temp*; *K* ←K_j ;
(*now we must move (*K*, *new*) and insert in parent internal node;
however, if parent is full, split may propagate*)
finished ←false;
repeat
if stack *S* is empty
then (*no parent node; new root node is created for the tree*)
begin
root ←a new empty internal node for the tree;
root ←<*n*, *K*, *new*>; finished ←true;
end
else **begin**
n ←pop stack *S*;
if internal node *n* is not full
then
begin (*parent node not full; no split*)
insert (*K*, *new*) in correct position in internal node *n*;
finished ←true
end
else **begin** (*internal node *n* is full with *p* tree pointers;
overflow condition; node is split*)copy *n* to *temp* (**temp* is an oversize internal node*);
insert (*K*, *new*) in *temp* in correct position;
(**temp* now has *p* + 1 tree pointers*)
new ←a new empty internal node for the tree;
j ← ⌊((*p* + 1)/2⌋;

n ←entries up to tree pointer P_j in *temp*;
(**n* contains <P_1, K_1, P_2, K_2, ..., P_{j-1}, K_{j-1}, P_j >*)
new ←entries from tree pointer P_{j+1} in *temp*;
(**new* contains < P_{j+1}, K_{j+1}, ..., K_{p-1}, P_p, K_p, P_{p+1} >*)
K ←K_j
(*now we must move (*K*, *new*) and insert in parent internal node*)
end
end
until finished
end;
end;

Figure 6.26 illustrates insertion of records in a B+-tree of order p = 3 and p_{leaf} = 2. First, we observe that the root is the only node in the tree, so it is also a leaf node. As soon as more than one level is created, the tree is divided into internal nodes and leaf nodes. Notice that *every key value must exist at the leaf level*, because all data pointers are at the leaf level. However, only some values exist in internal nodes to guide the search. Notice also that every value appearing in an internal node also appears as *the rightmost value* in the leaf level of the subtree pointed at by the tree pointer to the left of the value. When a *leaf node* is full and a new entry is inserted there, the node *overflows* and must be split. The first j = ⌈(($p$leaf + 1)/2)⌉ entries in the original node

are kept there, and the remaining entries are moved to a new leaf node. The *j*th search value is replicated in the parent internal node, and an extra pointer to the new node is created in the parent. These must be inserted in the parent node

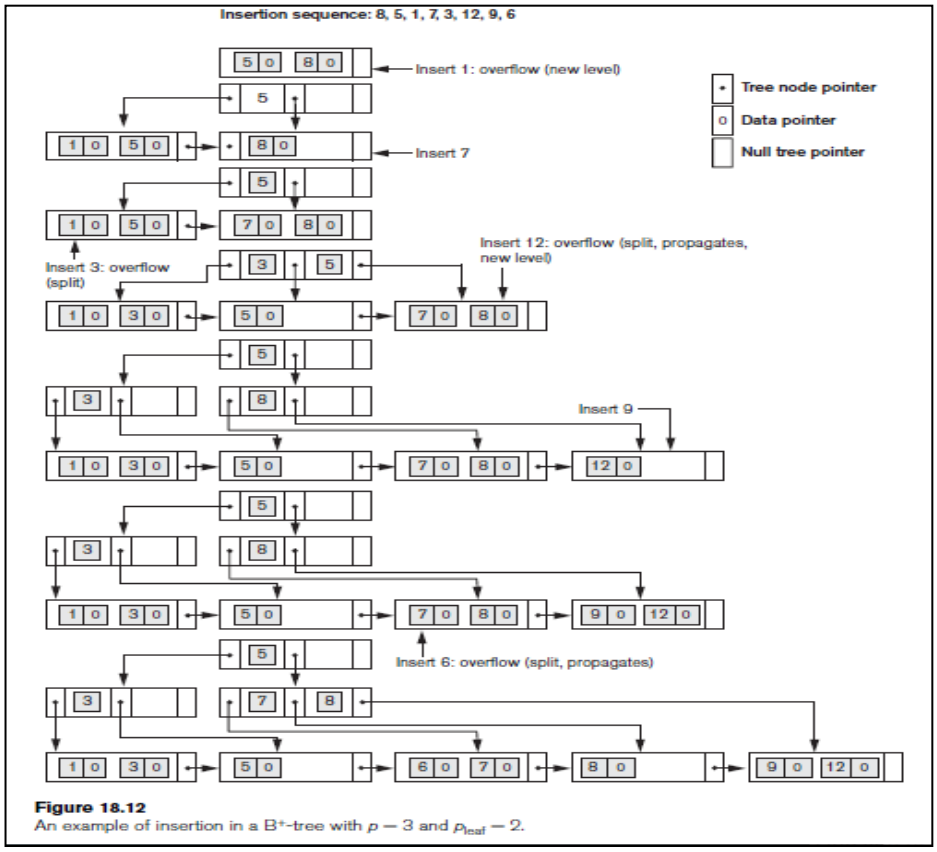


Figure 6.26: Insertion in B+ tree

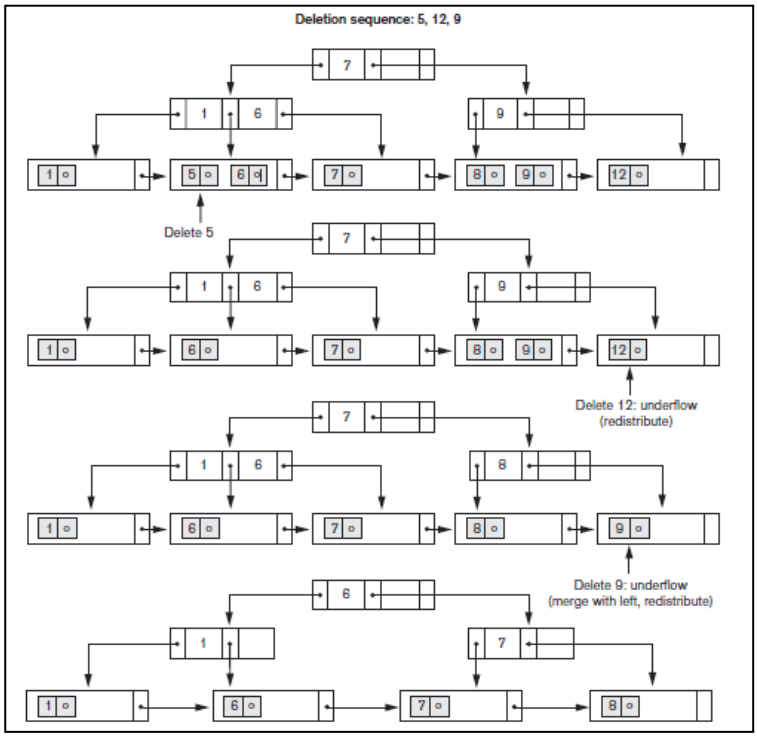


Figure 6.27: Deletion from B+ tree

in their correct sequence. If the parent internal node is full, the new value will cause it to overflow also, so it must be split. The entries in the internal node up to P_j—the jth tree pointer after inserting the new value and pointer, where $j = \lfloor((p + 1)/2)\rfloor$—are kept, while the jth search value is moved to the parent, not replicated. A new internal node will hold the entries from P_{j+1} to the end of the entries in the node (see Algorithm 6.6). This splitting can propagate all the way up to create a new root node and hence a new level for the B+-tree. Figure 6.27 illustrates deletion from a B+-tree. When an entry is deleted, it is always removed from the leaf level. If it happens to occur in an internal node, it must also be removed from there. In the latter case, the value to its left in the leaf node must replace it in the internal node because that value is now the rightmost entry in the subtree. Deletion may cause **underflow** by reducing the number of entries in the leaf node to below the minimum required. In this case, we try to find a sibling leaf node—a leaf node directly to the left or to the right of the node with underflow—and redistribute the entries among the node and its **sibling** so that both are at least half full; otherwise, the node is merged with its siblings and the number of leaf nodes is reduced. A common method is to try to **redistribute** entries with the left sibling; if this is not possible, an attempt to redistribute with the right sibling is such a case, underflow may propagate to **internal** nodes because one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.

Notice that implementing the insertion and deletion algorithms may require parent and sibling pointers for each node, or the use of a stack as in Algorithm 6.6. Each node should also include the number of entries in it and its type (leaf or internal). Another alternative is to implement insertion and deletion as recursive procedures.

Variations of B-Trees and B+-Trees. To conclude this section, we briefly mention some variations of B-trees and B+-trees. In some cases, constraint 5 on the Btree (or for the internal nodes of the B+–tree, except the root node), which requires each node to be at least half full, can be changed to require each node to be at least two-thirds full. In this case the B-tree has been called a **B*-tree**. In general, some systems allow the user to choose a **fill factor** between 0.5 and 1.0, where the latter means that the B-tree (index) nodes are to be completely full. It is also possible to specify two fill factors for a B+-tree: one for the leaf level and one for the internal nodes of the tree. When the index is first constructed, each node is filled up to approximately the fill factors specified. Some investigators have suggested relaxing the requirement that a node be half full, and instead allow a node to become completely empty before merging, to simplify the deletion algorithm. Simulation studies show that this does not waste too much additional space under randomly distributed insertions and deletions.

6.15 Indexes on Multiple Keys

In our discussion so far, we have assumed that the primary or secondary keys on which files were accessed were single attributes (fields). In many retrieval and update requests, multiple attributes are involved. If a certain combination of attributes is used frequently, it is advantageous to set up an access structure to provide efficient access by a key value that is a combination of those attributes.

For example, consider an EMPLOYEE file containing attributes Dno (department number), Age, Street, City, Zip_code, Salary and Skill_code, with the key of Ssn (Social Security number). Consider the query: *List the employees in department number 4 whose age is 59*.Note that both Dno and Age are nonkey attributes, which means that a search value for either of these will point to multiple records. The following alternative search strategies may be considered:

1. Assuming Dno has an index, but Age does not, access the records having Dno = 4 using the index, and then select from among them those records that satisfy Age = 592. Alternately, if Age is indexed but Dno is not, access the records having Age = 59 using the index, and then select from among them those records that satisfy Dno = 4.
3. If indexes have been created on both Dno and Age, both indexes may be used; each gives a set of records or a set of pointers (to blocks or records).An intersection of these sets of records or pointers yields those records or pointers that satisfy both conditions. All of these alternatives eventually give the correct result. However, if the set of records that meet each condition (Dno = 4 or Age = 59) individually are large, yet only a few records satisfy the combined condition, then none of the above is an efficient technique for the given search request. A number of possibilities

exist that would treat the combination < Dno, Age> or < Age, Dno> as a search key made up of multiple attributes. We briefly outline these techniques in the following sections. We will refer to keys containing multiple attributes as **composite keys**.

6.15.1 Ordered Index on Multiple Attributes

All the discussion in this chapter so far still applies if we create an index on a search key field that is a combination of <Dno, Age>. The search key is a pair of values <4, 59> in the above example. In general, if an index is created on attributes $<A_1, A_2, ..., A_n>$, the search key values are tuples with n values: $<v_1, v_2, ..., v_n>$. A lexicographic ordering of these tuple values establishes an order on this composite search key. For our example, all of the department keys for department number 3 precede those for department number 4. Thus <3, n> precedes <4,m> for any values of m and n. The ascending key order for keys with Dno = 4 would be <4, 18>, <4, 19>, <4, 20>, and so on. Lexicographic ordering works similarly to ordering of character strings. An index on a composite key of n attributes works similarly to any index discussed in this chapter so far.

6.15.2 Partitioned Hashing

Partitioned hashing is an extension of static external hashing (Section 17.8.2) that allows access on multiple keys. It is suitable only for equality comparisons; range queries are not supported. In partitioned hashing, for a key consisting of n components, the hash function is designed to produce a result with n separate hash addresses. The bucket address is a concatenation of these n addresses. It is then possible to search for the required composite search key by looking up the appropriate buckets that match the parts of the address in which we are interested.

For example, consider the composite search key <Dno, Age>. If Dno and Age are hashed into a 3-bit and 5-bit address respectively, we get an 8-bit bucket address. Suppose that Dno = 4 has a hash address ‘100’ and Age = 59 has hash address ‘10101’. Then to search for the combined search value, Dno = 4 and Age = 59, one goes to bucket address 100 10101; just to search for all employees with Age = 59, all buckets

(eight of them) will be searched whose addresses are ‘000 10101’, ‘001 10101’, ... and so on. An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high-order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be maintained for the individual attributes. The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes.

6.15.3 Grid Files

Another alternative is to organize the EMPLOYEE file as a grid file. If we want to access a file on two keys, say Dno and Age as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes. Figure 6.28 shows a grid array for the EMPLOYEE file with one linear scale for Dno and another for the Age attribute. The scales are made in a way as to achieve a uniform distribution of that attribute. Thus, in our example, we show that the linear scale for Dno has Dno = 1, 2 combined as one value 0 on the scale, while Dno = 5 corresponds to the value 2 on that scale. Similarly, Age is divided into its scale of 0 to 5 by grouping ages so as to distribute the employees uniformly by age. The grid array shown for this file has a total of 36 cells. Each cell points to some bucket address where the records corresponding to that cell are stored. Figure 6.28 also shows the assignment of cells to buckets (only partially). Thus our request for Dno = 4 and Age = 59 maps into the cell (1, 5) corresponding to the grid array. The records for this combination will be found in the corresponding bucket. This method is particularly useful for range queries that would map into a set of cells corresponding to a group of values along the linear scales. If a range query corresponds to a match on the some of the grid cells, it can be processed by accessing exactly the buckets for those grid cells. For example, a query for Dno ≤5and Age > 40 refers to the data in the top bucket shown in Figure 6.28. The grid file concept can be applied to any number of search keys. For example, for n search keys, the grid array would have n dimensions. The grid array thus allows a partitioning of the file along the dimensions of the search key attributes and provides an access by combinations of values along those dimensions. Grid files perform well in terms of reduction in time for multiple key access. However, they represent a space overhead in

terms of the grid array structure. Moreover, with dynamic files, a frequent reorganization of the file adds to the maintenance cost.

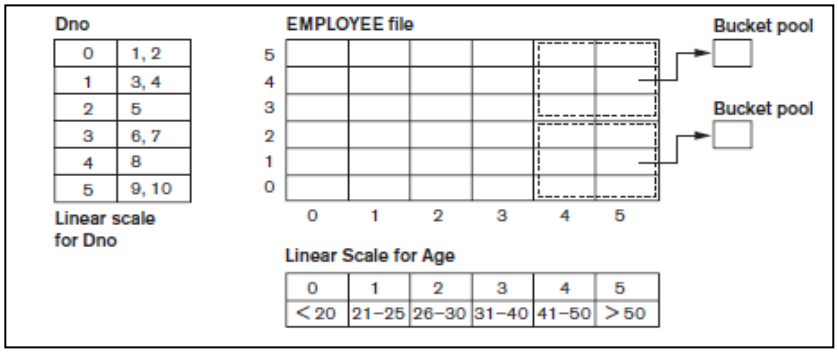


Figure 6.28: Sample Grid file

6.16 Other Types of Indexes

6.16.1 Hash Indexes

It is also possible to create access structures similar to indexes that are based on *hashing*. The **hash index** is a secondary structure to access the file by using hashing on a search key other than the one used for the primary data file organization. The index entries are of the type <*K*, *Pr*> or <*K*, *P*>, where *Pr* is a pointer to the record containing the key, or *P* is a pointer to the block containing the record for that key.

The index file with these index entries can be organized as a dynamically expandable hash file, using one of the techniques described in Section 17.8.3; searching for an entry uses the hash search algorithm on *K*. Once an entry is found, the pointer *Pr* (or *P*) is used to locate the corresponding record in the data file. Figure 6.28 illustrates a hash index on the Emp_id field for a file that has been stored as a sequential file ordered by Name. The Emp_id is hashed to a bucket number by using a hashing function: the sum of the digits of Emp_id modulo 10. For example, to find Emp_id 51024, the hash function results in bucket number 2; that bucket is accessed first. It contains the index entry < 51024, *Pr* >; the pointer *Pr* leads us to the actual record in the file. In a practical application, there may be thousands of buckets; the bucket number, which may be several bits long, would be subjected to the directory schemes discussed about dynamic hashing in Section 17.8.3. Other search structures can also be used as indexes.

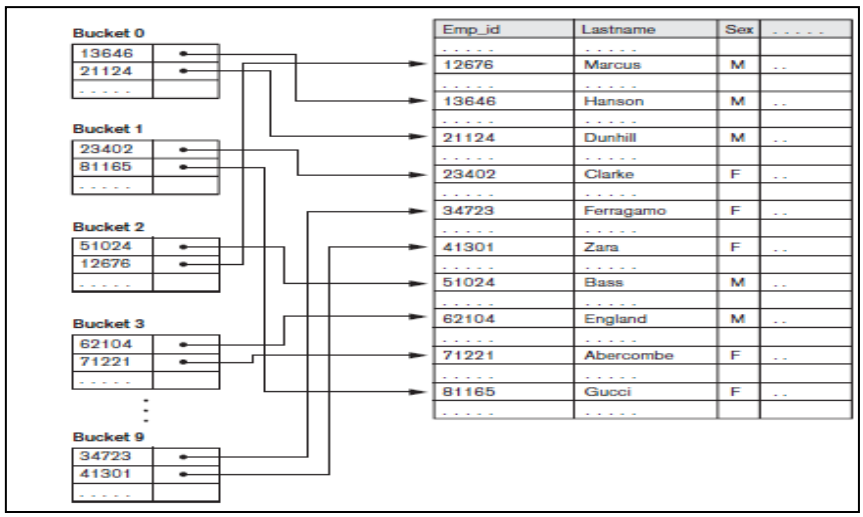
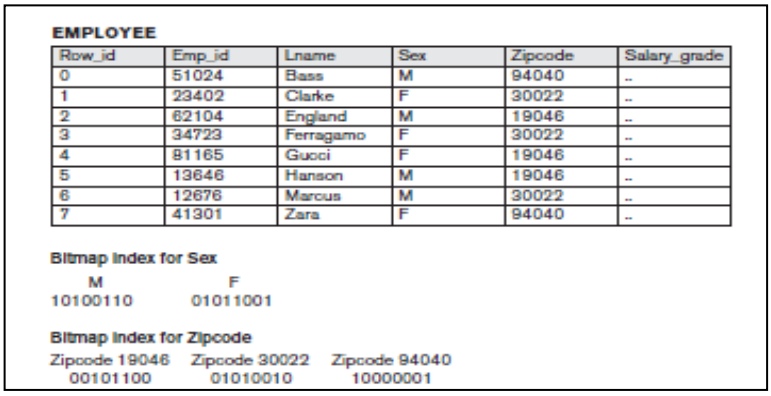


Figure 6.29: Sample Hash index

6.16.2 Bitmap Indexes

The **bitmap index** is another popular data structure that facilitates querying on multiple keys. Bitmap indexing is used for relations that contain a large number of rows. It creates an index for one or more columns, and each value or value range in those columns is indexed. Typically, a bitmap index is created for those columns that contain a fairly small number of unique values. To build a bitmap index on a set of records in a relation, the records must be numbered from 0 to *n* with an id (a record id or a row id) that can be mapped to a physical address made of a block number and a record offset within the block. A bitmap index is built on one particular value of a particular field (the column in a relation) and is just an array of bits. Consider a bitmap index for the column *C* and a value *V* for that column. For a relation with *n* rows, it contains *n* bits. The *i*th bit is set to 1 if the row *i* has the value *V* for column *C*; otherwise it is set to a 0. If *C* contains the valueset $<v_1, v_2, ..., v_m>$ with *m* distinct values, then *m* bitmap indexes would be created for that column. Figure 6.30 shows the relation EMPLOYEE with columns Emp_id, Lname, Sex, Zipcode, and Salary_grade (with just 8 rows for illustration) and a bitmap index for the Sex and Zipcode columns. As an example, if the bitmap for Sex = F, the bits for Row_ids 1, 3, 4, and 7 are set to 1, and the rest of the bits are set to 0, the bitmap indexes could have the following query applications:



EMPLOYEE

Row_id	Emp_id	Lname	Sex	Zipcode	Salary_grade
0	51024	Bass	M	94040	..
1	23402	Clarke	F	30022	..
2	62104	England	M	19046	..
3	34723	Ferragamo	F	30022	..
4	81165	Gucci	F	19046	..
5	13646	Hanson	M	19046	..
6	12676	Marcus	M	30022	..
7	41301	Zara	F	94040	..

Figure 6.30: Sample Bitmap index

For the query $C_1 = V_1$, the corresponding bitmap for value V_1 returns the Row_ids containing the rows that qualify.

■ For the query $C_1 = V_1$ and $C_2 = V_2$ (a multikey search request), the two corresponding bitmaps are retrieved and intersected (logically AND-ed) to yield the set of Row_ids that qualify. In general, *k* bitvectors can be intersected to deal with *k* equality conditions. Complex AND-OR conditions can also be supported using bitmap indexing.

■ To retrieve a count of rows that qualify for the condition $C_1 = V_1$, the "1" entries in the corresponding bitvector are counted.

■ Queries with negation, such as $C_1 \neg = V_1$, can be handled by applying the Boolean *complement* operation on the corresponding bitmap.

Consider the example in Figure 6.30. To find employees with Sex = F and Zipcode = 30022, we intersect the bitmaps "01011001" and "01010010" yielding Row_ids 1 and 3. Employees who do not live in Zipcode = 94040 are obtained by complementing the bitvector "10000001" and yields Row_ids 1 through 6. In general, if we assume uniform distribution of values for a given column, and if one column has 5 distinct values and another has 10 distinct values, the join condition on these two can be considered to have a selectivity of 1/50 (=1/5 * 1/10). Hence, only about 2 percent of the records would actually have to be retrieved. If a column has only a few values, like the Sex column in Figure 6.30, retrieval of the Sex = M condition on average would retrieve 50 percent of the rows; in such cases, it is better to do a complete scan rather than use bitmap indexing. In general, bitmap indexes are efficient in terms of the storage space that they need.

If we consider a file of 1 million rows (records) with record size of 100 bytes per row, each bitmap index would take up only one bit per row and hence would use 1 million bits or 125 Kbytes. Suppose this relation is for 1 million residents of a state, and they are spread over 200 ZIP Codes; the 200 bitmaps over Zipcodes contribute 200 bits (or 25 bytes) worth of space per

row; hence, the 200 bitmaps occupy only 25 percent as much space as the data file. They allow an exact retrieval of all residents who live in a given ZIP Code by yielding their Row_ids. When records are deleted, renumbering rows and shifting bits in bitmaps becomes expensive. Another bitmap, called the **existence bitmap**, can be used to avoid this expense. This bitmap has a 0 bit for the rows that have been deleted but are still present and a 1 bit for rows that actually exist. Whenever a row is inserted in the relation, an entry must be made in all the bitmaps of all the columns that have a bitmap index; rows typically are appended to the relation or may replace deleted rows. This process represents an indexing overhead.

Large bit vectors are handled by treating them as a series of 32-bit or 64-bit vectors, and corresponding AND, OR, and NOT operators are used from the instruction set to deal with 32- or 64-bit input vectors in a single instruction. This makes bit vector operations computationally very efficient.

Bitmaps for B+-Tree Leaf Nodes. Bitmaps can be used on the leaf nodes of B+-tree indexes as well as to point to the set of records that contain each specific value of the indexed field in the leaf node. When the B+-tree is built on a nonkey search field, the leaf record must contain a list of record pointers alongside each value of the indexed attribute. For values that occur very frequently, that is, in a large percentage of the relation, a bitmap index may be stored instead of the pointers.

As an example, for a relation with *n* rows, suppose a value occurs in 10 percent of the file records. A bit vector would have *n* bits, having the "1" bit for those Row_ids that contain that search value, which is *n*/8 or 0.125*n* bytes in size. If the record pointer takes up 4 bytes (32 bits), then the *n*/10 record pointers would take up 4 * *n*/10 or 0.4*n* bytes. Since 0.4*n* is more than 3 times larger than 0.125*n*, it is better to store the bitmap index rather than the record pointers. Hence for search values that occur more frequently than a certain ratio (in this case that would be 1/32), it is beneficial to use bitmaps as a compressed storage mechanism for representing the record pointers in B+-trees that index a nonkey field.

6.16.3 Function-Based Indexing

In this section we discuss a new type of indexing, called **function-based indexing**, that has been introduced in the Oracle relational DBMS as well as in some other commercial products. The idea behind function-based indexing is to create an index such that the value that results from applying some function on a field or a collection of fields becomes the key to the index. The following examples show how to create and use function based indexes.

Example 1. The following statement creates a function-based index on the EMPLOYEE table based on an uppercase representation of the Lname column, which can be entered in many ways but is always queried by its uppercase representation.

CREATE INDEX upper_ix ON Employee (UPPER(Lname)); This statement will create an index based on the function UPPER(Lname), which returns the last name in uppercase letters; for example, UPPER('Smith') will return 'SMITH'.

Function-based indexes ensure that Oracle Database system will use the index rather than perform a full table scan, even when a function is used in the search predicate of a query. For example, the following query will use the index:

```
SELECT First_name, Lname
FROM Employee
WHERE UPPER(Lname)= "SMITH".
```

Without the function-based index, an Oracle Database might perform a full table scan, since a B+-tree index is searched only by using the column value directly; the use of any function on a column prevents such an index from being used.

Example 2. In this example, the EMPLOYEE table is supposed to contain two fields—salary and commission_pct (commission percentage)—and an index is being created on the sum of salary and commission based on the commission_pct.

```
CREATE INDEX income_ix
ON Employee(Salary + (Salary*Commission_pct));
```

The following query uses the income_ix index even though the fields salary and commission_pct are occurring in the reverse order in the query when compared to the index definition.

```
SELECT First_name, Lname
FROM Employee
WHERE ((Salary*Commission_pct) + Salary ) > 15000;
```

Example 3. This is a more advanced example of using function-based indexing to define conditional uniqueness. The following statement creates a unique function based index on the ORDERS table that prevents a customer from taking advantage of a promotion id ("blowout sale") more than once. It creates a composite index on the Customer_id and Promotion_id fields together, and it allows only one entry in the index for a given Customer_id with the Promotion_id of "2" by declaring it as a unique index.

```
CREATE UNIQUE INDEX promo_ix ON Orders
(CASE WHEN Promotion_id = 2 THEN Customer_id ELSE NULL END,
CASE WHEN Promotion_id = 2 THEN Promotion_id ELSE NULL END);
```

Note that by using the **CASE** statement, the objective is to remove from the index any rows where Promotion_id is not equal to 2. Oracle Database does not store in the B+- tree index any rows where all the keys are NULL. Therefore, in this example, we map both Customer_id and Promotion_id to NULL unless Promotion_id is equal to 2. The result is that the index constraint is violated only if Promotion_id is equal to 2, for two (attempted insertions of) rows with the same Customer_id value.

6.16.3 Column-Based Storage of Relations

There has been a recent trend to consider a column-based storage of relations as an alternative to the traditional way of storing relations row by row. Commercial relational DBMSs have offered B+-tree indexing on primary as well as secondary keys as an efficient mechanism to support access to data by various search criteria and the ability to write a row or a set of rows to disk at a time to produce write-optimized systems. For data warehouses (to be discussed in Chapter 29), which are read-only databases, the column-based storage offers particular advantages for read-only queries. Typically, the column-store RDBMSs consider storing each column of data individually and afford performance advantages in the following areas:

- Vertically partitioning the table column by column, so that a two-column table can be constructed for every attribute and thus only the needed columns can be accessed
- Use of column-wise indexes (similar to the bitmap indexes) and join indexes on multiple tables to answer queries without having to access the data tables.
- Use of materialized views to support queries on multiple columns. Column-wise storage of data affords additional freedom in the creation of indexes, such as the bitmap indexes discussed earlier. The same column may be present in multiple projections of a table and indexes may be created on each projection. To store the values in the same column, strategies for data compression, null-value suppression, dictionary encoding techniques (where distinct values in the column are assigned shorter codes), and run-length encoding techniques have been devised. MonetDB/X100, C-Store, and Vertica are examples of such systems.

MODULE 6 MCQ and Short type Problem

Multiple choice questions (MCQ)

1 The fields which are used to retrieve the related records from other files are called

a) secondary fields
b) primary fields
c) key fields
d) connecting fields

Answer: (d)

2. The primary indexes, secondary indexes and cluster indexes are all types of

a) ordered indexes
b) unordered indexes
c) linear indexes
d) relative search indexes

Answer: (a)
3. B-tree eliminates the redundant storage of
a) Search keys
b) Indices
c) Buckets
d) Bucket skew
Answer: (a)
4. The nonleaf nodes of the B+- tree structure form a
a) Multilevel clustered indices
b) Sparse indices
c) Multilevel dense indices
d) Multilevel sparse indices
Answer: (d)
5. While insertion in a sparse index, we assume that the index stores an entry for each
a) Index
b) Pointer
c) Position
d) Block
Answer: (d)

Short question

1. What is the difference between primary and secondary storage?
2. Why is accessing a disk block expensive? Discuss the time components involved in accessing a disk block.
3. What is the difference between the directories of extendible and dynamic hashing?
4. What characterizes the levels in RAID organization?
5. How does disk mirroring help improve reliability? Give a quantitative example.
6. How does multilevel indexing improve the efficiency of searching an index file?
7. What is the order *p* of a B-tree? Describe the structure of B-tree nodes.
8. How does a B-tree differ from a B+-tree? Why is a B+-tree usually preferred as an access structure to a data file?
9. What is partitioned hashing? How does it work? What are its limitations?
10. What is a fully inverted file? What is an indexed sequential file?

Long type question

1. What are the highlights of the popular RAID levels 0, 1, and 5?
2. Discuss the advantages and disadvantages of using (a) an unordered file, (b) an ordered file, and (c) a static hash file with buckets and chaining. Which operations can be performed efficiently on each of these organizations, and which operations are expensive?
3. Discuss the techniques for allowing a hash file to expand and shrink dynamically. What are the advantages and disadvantages of each?
4. A file has r = 20,000 STUDENT records of *fixed length.* Each record has the following fields: Name (30 bytes), Ssn (9 bytes), Address (40 bytes), PHONE (10 bytes), Birth_date (8 bytes), Sex (1 byte), Major_dept_code (4 bytes), Minor_dept_code (4 bytes), Class_code (4 bytes, integer), and Degree_program (3 bytes).An additional byte is used as a deletion marker. The file is stored on the disk whose parameters are given in Exercise 6.27.
a. Calculate the record size R in bytes.
b. Calculate the blocking factor *bfr* and the number of file blocks b, assuming an unspanned organization.
c. Calculate the average time it takes to find a record by doing a linear search on the file if (i) the file blocks are stored contiguously, and double buffering is used; (ii) the file blocks are not stored contiguously.

d. Assume that the file is ordered by Ssn; by doing a binary search, calculate the time it takes to search for a record given its Ssn value.
5. A PARTS file with Part# as the hash key includes records with the following Part# values: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, and 9208. The file uses eight buckets, numbered 0 to 7. Each bucket is one disk block and holds two records. Load these records into the file in the given order, using the hash function $h(K) = K$ mod 8. Calculate the average number of block accesses for a random retrieval on Part#.
6. What are the differences among primary, secondary, and clustering indexes? How do these differences affect the ways in which these indexes are implemented? Which of the indexes are dense, and which are not?
7. A PARTS file with Part# as the key field includes records with the following Part# values: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38. Suppose that the search field values are inserted in the given order in a B+-tree of order $p = 4$ and $p_{leaf} = 3$; show how the tree will expand and what the final tree will look like.
8. Suppose that the following search field values are deleted, in the given order, from the B+-tree of Exercise 2; show how the tree will shrink and show the final tree. The deleted values are 65, 75, 43, 18, 20, 92, 59, 37.
9. Suppose that several secondary indexes exist on nonkey fields of a file, implemented using option 3; for example, we could have secondary indexes on the fields Department_code, Job_code, and Salary of the EMPLOYEE file of Exercise. Describe an efficient way to search for and retrieve records satisfying a complex selection condition on these fields, such as (Department_code = 5 AND Job_code = 12 AND Salary = 50,000), using the record pointers in the indirection level.
10. Consider a disk with block size $B = 512$ bytes. A block pointer is $P = 6$ bytes long, and a record pointer is $PR = 7$ bytes long. A file has $r = 30{,}000$ EMPLOYEE records of *fixed length*. Each record has the following fields: Name (30 bytes), Ssn (9 bytes), Department_code (9 bytes), Address (40 bytes), Phone (10 bytes), Birth_date (8 bytes), Sex (1 byte), Job_code (4 bytes), and Salary (4 bytes, real number).An additional byte is used as a deletion marker. a. Calculate the record size R in bytes. b. Calculate the blocking factor *bfr* and the number of file blocks *b*, assuming an unspanned organization.

Text Books:

1. Henry F. Korth and Silberschatz Abraham, “Database System Concepts”, Mc.Graw Hill.
2. Elmasri Ramez and Novathe Shamkant, “Fundamentals of Database Systems”, Benjamin Cummings Publishing. Company.
3. Date C. J., “Introduction to Database Management”, Vol. I, II, III, Addison Wesley.
4. Database Management Systems”, Arun K.Majumdar, Pritimay Bhattacharya, Tata McGraw Hill.